A Scalable and Parallel Inductive Learner in Description Logic

Eyad Algahtani

Doctor of Philosophy

University of York

Computer Science

October 2020

Abstract

Inductive logic programming (ILP) is a type of supervised machine learning where the data and the discovered models are described using logic formalisms. The use of such formalisms has strong expressive power at describing a variety of complex concepts and relations. However, when expressive logic formalisms, such as description logic (DL), are combined with large data, the ILP learning performance is greatly reduced. Consequently, real-world applications for ILP learning are limited to small or medium datasets.

In this work, we present SPILDL (a Scalable and Parallel Inductive Learner in DL). SPILDL builds upon the DL-Learner, the current state of the art for DL-based ILP learning. SPILDL exploits the parallel computing capabilities of multi-core CPUs to accelerate hypothesis exploration and multiple GPUs for hypothesis evaluation. The result is a scalable and high-performance DL-based ILP learner that can handle and cope directly with large and complex real-world datasets. In addition, a variation of SPILDL known as SPILDL-ES (SPILDL for Embedded Systems) is also proposed. SPILDL-ES is a high-performance DL-based ILP learner specifically designed to exploit parallel computing capabilities of embedded systems hardware. SPILDL-ES handles both ILP learning and monitoring (and controlling) of the embedded hardware in real time. The evaluations of SPILDL and SPILDL-ES on both synthetic and real-world datasets show that speedups up to 360 times faster than the original speed can be reached. It is anticipated this work contributes significantly to the research and development of DL-based ILP systems.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

Foremost, I would like to express my greatest gratitude to my sponsor, my country, the Kingdom of Saudi Arabia and my employer the King Saud University for giving me the opportunity to pursue a PhD degree and to support me financially.

I would like to thank my family especially my parents for their never-ending support and encouragement which greatly helped my PhD journey.

I would like to give my heartfelt gratitude to my supervisor Dr. Tommy Yuan for his supervision, never-ending guidance and support, and encouraging words that greatly helped my research. I have been working in the field of high-performance computing for many years before my PhD, where I have professionally developed (or accelerated) many complex data analysis algorithms using parallel computing technologies on variety of hardware and computing architectures for many domains, notably, decision support and healthcare. I am proud with the work I have achieved for my research under Dr. Yuan's supervision, in terms theoretical and technical aspects. I would like to thank Dr. Yuan for also supervising my thesis, where he always provided me with a fast and concise feedback which greatly improved my research and my research skills. I would like to thank Dr. James Cussens (a member of my thesis advisory panel) for giving me a very insightful feedback which greatly helped my research.

I would like to thank my PhD examiners Prof. Alessandra Russo and Dr. Peter Nightingale for their detailed feedback on my research, which made me a better researcher.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, university. All sources are acknowledged as References. In this thesis, some materials appeared in the following published papers.

Algahtani, E. and Kazakov, D. (2018) 'GPU-Accelerated Hypothesis Cover Set Testing for Learning in Logic', *Proceedings of the 28th International Conference on Inductive Logic Programming (ILP2018)*, Ferrara, September 2, pp. 6-20.

Algahtani, E. and Kazakov, D. (2019) 'CONNER: A Concurrent ILP Learner in Description Logic', *Proceedings of the 29th International Conference on Inductive Logic Programming (ILP2019)*, Plovdiv, September 3, pp. 1-15.

Chapter 1: Introduction

1.1 Background and rationale

Inductive logic programming (ILP) has been successfully used in many domains to construct white-box (i.e. human readable) models for both description and prediction. Inductive logic programming applications encompass a wide variety of domains, including for predicting chemical structures in biochemistry (Debnath *et al.*, 1991), learning diagnostic rules for satellite temporal faults (Feng, 1991) in aerospace engineering, and many others. Indeed, ILP learning is not bound to single logic formalism. In fact, even though ILP is traditionally used with first-order logic (horn clauses in particular), it has demonstrated its success with other logic formalisms, such as in description logic (DL). For example, in recommender systems applications (Qomariyah and Kazakov, 2017), user preferences are learned (directly) from ordinal data represented in DL.

Additionally, ILP learning in DL requires less computational power and can handle more data than horn clauses. However, DL is a less expressive logic formalism than horn clauses. As a machine learning technique, ILP is still computationally expensive because of its ability to directly handle the complexity and size of multi-relational data. Generally, in ILP learning, the more expressive the logic formalism is, the more (computationally) expensive is the ILP learning. With this in mind, unlike horn clauses, DL provides a trade-off between expressivity and computational complexity, which enables DL-based ILP learners to learn more expressive models than propositional logic, while requiring less computing resources than first order logic; this trade-off places DL (in terms of expressivity) between propositional logic and first order logic. As a result, DL is suitable for many ILP learning cases, such as the DL-Learner (Lehmann, 2010).

Many of ILP learners (algorithms) formulate the ILP learning as a search problem, exploring the search space of possible models (or solutions) for an appropriate solution. The complexity of the search space is mainly determined by the logic formalism employed, which grows in relation to the expressive power of the logic formalism. In addition, when more learning examples are used, the time spent in evaluating a single search node (or a candidate solution) increases. These issues pose key challenges in the application of ILP learning to real-world datasets, which are complex, large, and in some cases, massive.

Many approaches have been proposed to deal with general ILP performance, such as sampling techniques (Srinivasan, 1999) to address the scalability problem for large datasets. In addition, some approaches, such as Query Packs (Blockeel *et al.*, 2000), have been developed to optimize the hypotheses evaluation, which aims to reduce unnecessary computations by reusing computation results. Moreover, some optimizations exist as part of an ILP learner – for example, mode declarations in Progol (Muggleton, 1995), which reduce (or prune) the search space.

Many of these techniques have managed to mitigate the scalability problem to some degree. However, there are some drawbacks. For example, sampling techniques may reduce the number of the learning examples but may also increase the risk of omitting crucial learning examples and as a result affect the quality of ILP learning. In Query Packs, hypothesis evaluation is optimized by reusing computation results, definitely improving performance. However, Query Packs only mitigates the scalability problem and does not directly address it. Mode declarations control Progol's hypothesis language, which can be used to reduce the potentially large search into a more manageable one; this reduction improves the performance of the hypothesis search. However, mode declarations require careful configuration to achieve desired results.

The computational needs of ILP, combined with its scalability problem, render its use in more computationally restricted environments, such as embedded systems or mobile applications, to be very limited or potentially non-existent. In other words, addressing the scalability and performance challenges associated with ILP will consequently enable the use of ILP learning in computationally limited hardware, such as embedded systems and mobile applications.

To address the aforementioned issues, we sought a scalable ILP learning algorithm for inductive learning in DL. The choice of the DL logic formalism was made for many reasons. First, the application of description logic has been commonly associated with semantic web technologies,[1] which are witnessing growing interest given the abundance of the available knowledge (e.g. Linked Open Data,[2] DBpedia,[3] and many others), thus expanding the horizon of potential ILP learning opportunities. In addition, the choice of description logic is also justified through its good compromise of expressivity and computational needs, as discussed above.

Parallel computing carries great general potential for accelerating computations, where multiple processors can work together simultaneously to solve a specific problem, such using a GPU for computing and rendering complex 3D graphics. In other words, parallel computing helps to solve computing problems more quickly by dedicating more computing power to those problems through multiple simultaneous processors. With these capabilities in mind, proper exploitation of parallel computing is expected to address the scalability problem of ILP learning for parallel ILPs in horn clauses (Nishiyama and Ohwada, 2015; Martínez-Angeles *et al.*, 2016), and parallel reasoning and hypothesis evaluation in description logic (Wu and Haarslev, 2012; Chantrapornchai and Choksuchat, 2018).

[1] https://www.w3.org/standards/semanticweb/
[2] https://www.w3.org/DesignIssues/LinkedData.html
[3] https://wiki.dbpedia.org/

As a result, we seek an inductive learner in DL, one which exploits the parallel computing capabilities of CPUs and GPUs, to address the scalability issue of ILP learning in DL. This learner must be able to explore large search spaces and evaluate hypotheses against large amount of data.

1.2 Research aim and objectives

The aim of this project was to address the scalability issues associated with ILP learning in description logic via the exploitation of parallel computing capabilities. In order to achieve the overall aim of this project, the following objectives were developed:

- Conduct a critical review of the parallel ILP literature.
- In light of the literature review, select or develop a new ILP algorithm (inspired by classic ILPs) to address ILP scalability issues.
- Seek suitable means to implement and evaluate the proposed solution.
- Study the performance of the solution in real-world scenarios, such as mobile and embedded systems applications.

This research answered the following research questions and their related hypotheses:

RQ1: How can scalability issues of DL-based ILP learning be addressed using parallel computing?

H_1: Parallel computing reduces hypothesis evaluation time for large DL datasets.

H_2: Parallel computing increases the speed at which large search spaces can be explored.

H_3: Parallel computing improves ILP performance in general, even on smaller datasets.

RQ2: To what degree can parallel computing improve ILP learning in DL?

H_4: Adding more parallel processors reduces ILP learning time.

H_5: Combining multiple types of processors (e.g. GPUs and CPUs) achieves further scalability improvements.

The two research questions are addressed in chapter 5, chapter 6, and chapter 7. In terms of research hypotheses, the testing of the first hypothesis is reported in chapter 5. Chapter 6 reports the testing of the second and third hypotheses. The testing of the third hypothesis is reported in chapter 7. The three aforementioned chapters also report the testing of the fourth and fifth hypotheses.

It was anticipated that answering the research questions would open more frontier for ILP as a large-scale machine learning technique.

1.3 Thesis overview

The remainder of the thesis is organized as follows. Chapter 2 reviews the fundamental concepts in ILP, DL, and parallel computing that are used throughout this work. Chapter 3 provides a review of inductive

learning (i.e. ILP learning) in the context of DL (as its logic formalism); it also discusses the state of the art in DL-based ILP learning (which this work is built on). Thereafter, in chapter 4, a critical analysis of the relevant (multidisciplinary) literature in ILP and DL parallel strategies is provided. After analysing and reviewing the literature, we discuss and evaluate in chapter 5 the proposed multi-GPU evaluation engine, which accelerates the hypothesis evaluation with multiple GPUs. Chapter 6 then describes how the evaluation engine was integrated with a parallel search algorithm to form the complete scalable DL-based learner. Chapter 6 provides the complete details of the design, implementation, and evaluation of the proposed learner. In chapter 7, we explore a variation of the proposed learner, specifically designed for embedded systems. Finally, chapter 8 concludes the thesis and provides avenues for future work.

Chapter 2: Fundamental concepts

In this chapter, we introduce the fundamental concepts in the relevant research areas that are necessary to understand this work. The research encompasses intersections of inductive logic programming, description logic, and parallel computing, as shown in Figure 2.1.

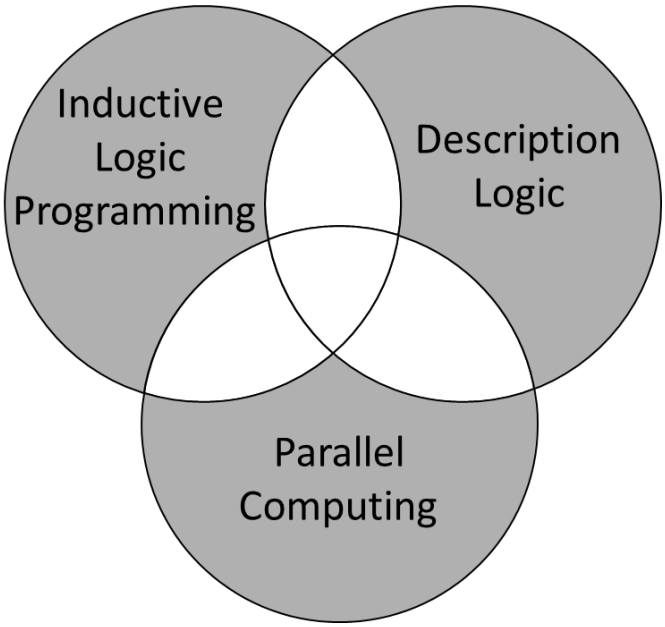


Figure 2.1: Relevant research areas.

The chapter starts with a review of fundamental concepts in ILP, which covers the following: basic ILP definitions, its learning process (as a machine learning technique), its various learning strategies (or approaches to achieve learning), its types of knowledge representations, its possible combinations with other (machine learning) techniques, and finally a brief discussion of ILP optimizations. It then provides a review of basic concepts in description logic, covering the following: the definition of description logic, knowledge representation with DL, its different variations (or languages), its knowledge representation constructors (or building blocks), its reasoning process (and a brief discussion of common DL reasoners), its relation to semantic web technologies, and finally its role in scalable (logic-based) knowledge representation (in the context of a semantic web). The chapter ends by providing a review of basic concepts in parallel computing: the definition of parallel computing, its computing architectures, its memory (architecture) types, a brief discussion of its computing domains (including common software APIs), a discussion of GPUs (GPGPUs in particular), and finally a discussion of (shared memory) multi (CPU) processing through the OpenMP API. The review of these three relevant research areas provides the necessary background to understand this work (and its multi-disciplinary nature).

2.1 Inductive Logic Programming

In this section, we provide a review of the fundamental concepts in ILP. ILP is a machine-learning technique that employs logic formalisms as a standard model of knowledge representation. In essence, ILP algorithms use training examples and background knowledge, which are represented in logic, to learn models that are expressed as a logic rule or as a set of logic rules; these logic-based models can be used for either description or predication purposes. In this context, ILP learning, as a type of inductive reasoning, has been used for concept learning in many domains, including engineering and biochemistry (Bratko and Muggleton, 1995).

2.1.1 ILP learning

ILP learning employs logic formalism as its means to represent knowledge and models. This type of learning makes use of training examples and the background knowledge. The training examples (*E*) are represented as a set of logic facts (or statements) containing the positive examples (E^+) and (sometimes) negative examples (E^-). In ideal situations, the ILP learning will result in a model (logic rule) that covers all the positive examples (complete) and none of the negative examples (consistent). During ILP learning, the training examples are used in conjunction with background knowledge (*B*), which contains various details (logic statements) about the training examples. In addition, any ILP learner has a hypothesis language L_H, which determines its building blocks (or constructors) used to build a hypothesis *H*. The ILP learning process can be demonstrated through the example of learning the logic-based description of a bird. In this ILP example, the positive examples will be the set of two birds: $creatureA$ (an eagle) and $creatureB$ (a chicken).

$$E^+ = \begin{cases} bird(creatureA) \leftarrow \\ bird(creatureB) \leftarrow \end{cases}$$

The negative examples would be a set of three other animals: $creatureC$ (a bat), $creatureD$ (a turtle), and $creatureE$ (a cat).

$$E^- = \begin{cases} \leftarrow bird(creatureC) \\ \leftarrow bird(creatureD) \\ \leftarrow bird(creatureE) \end{cases}$$

The background knowledge will include details about the learning examples (both positive and negative), such as the existence of wings and beaks, type of wings (feather or membrane), and other details.

$$B = \begin{cases} animal(creatureA) \leftarrow \\ hasBeak(creatureA) \leftarrow \\ hasTalons(creatureA) \leftarrow \\ hasWing(creatureA, wingA1) \leftarrow \\ hasWing(creatureA, wingA2) \leftarrow \\ feather(wingA1) \leftarrow \\ feather(wingA2) \leftarrow \\ animal(creatureB) \leftarrow \\ hasBeak(creatureB) \leftarrow \\ hasClaws(creatureB) \leftarrow \\ hasWing(creatureB, wingB1) \leftarrow \\ hasWing(creatureB, wingB2) \leftarrow \\ feather(wingB1) \leftarrow \\ feather(wingB2) \leftarrow \\ animal(creatureC) \leftarrow \\ hasTeeth(creatureC) \leftarrow \\ hasWing(creatureC, wingC1) \leftarrow \\ hasWing(creatureC, wingC2) \leftarrow \\ membrane(wingC1) \leftarrow \\ membrane(wingC2) \leftarrow \\ animal(creatureD) \leftarrow \\ hasBeak(creatureD) \leftarrow \\ hasFlippers(creatureD) \leftarrow \\ animal(creatureE) \leftarrow \\ hasSharpTeeth(creatureE) \leftarrow \\ hasClaws(creatureE) \leftarrow \\ hasWhiskers(creatureE) \leftarrow \\ hasTeeth(X) \leftarrow hasSharpTeeth(X) \\ carnivore(X) \leftarrow hasSharpTeeth(X); hasTalons(X) \end{cases}$$

The ILP learner uses E and the details in B to inductively infer a hypothesis $H \in L_H$:

$$H = bird(X) \leftarrow animal(X), hasWing(X, Y), feather(Y)$$

This hypothesis describes a bird as an animal that has wings covered in feathers. The hypothesis H is complete since $\forall e \in E^+ : B \wedge H \vDash e$ and consistent since $\forall e \in E^- : B \wedge H \nvDash e$. In many ILP learning algorithms, the learning is formulated as a search problem (i.e. a search algorithm is used), where many search nodes (candidate hypotheses) are generated and then evaluated based on certain criteria; the hypothesis that satisfies the desired criteria is the learned model. The candidate hypotheses are generated by the refinement operator.

The refinement operator generates hypotheses through L_H, and these generated hypotheses can be either completely new or variations of existing ones. There are two types of refinements: general-to-specific (top-down), and specific-to-general (bottom-up). In a general-to-specific refinement, an overgeneralized hypothesis is generated first, which covers all positive examples and potentially some negative examples. The overly general hypothesis is then refined, whereby child hypotheses are generated; these generated hypotheses are more restricted (specialized) than their parent hypotheses

because they have more conditions that need to be satisfied. Ideally, the more specialized hypotheses should still cover all the positive examples while at the same time reducing the covered negatives by adding more restrictions. For the specific-to-general refinement operators, a positive learning example is picked from the set of positive learning examples (which covers zero negative examples by default). Thereafter, this learning example is refined by generating child hypotheses that are more general than the parent hypothesis. Ideally, these generalized hypotheses should still cover the same positive examples as their parent hypotheses or more while at the same time maintaining zero or minimal covered negatives.

Refinement operators have several characteristics: completeness, properness, finiteness, and ideal. A refinement operator is complete if it can reach all generalizations (in case of upward refining) or all specializations (in case of downward refining) of a given hypothesis through a finite set of refinement steps. A proper refinement operator generates child refinements (hypotheses) that are not equal to their parent ones. A refinement operator is finite if it generates a finite set of refinements for a given hypothesis.

A complete, proper, and finite refinement operator is known as "ideal". Ideal refinement operators are difficult to design, especially for expressive logics.

2.1.2 ILP learning strategies

When ILP is formulated as a search problem, there are three strategies: top-down, bottom-up, and hybrid. In order to demonstrate the strategies in action, the bird example from above is used.

Top-down strategy

In this strategy, an ILP learner uses a general-to-specific refinement operator. An example of learning the description of a bird using this strategy can be seen in Figure 2.2.

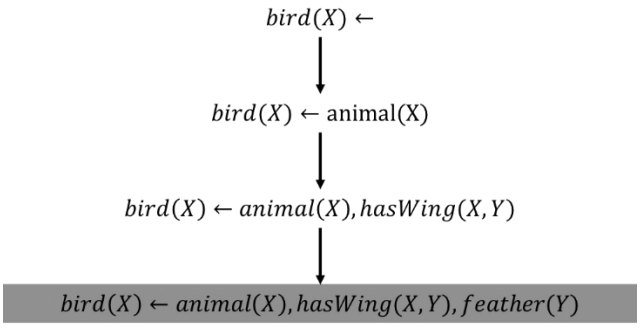


Figure 2.2: Learning the description of a bird using top-down strategy.

In Figure 2.2, the learning starts first at the most general hypothesis $bird(X) \leftarrow$, which covers everything and always returns true. Second, this hypothesis is then restricted by adding $animal(X)$, which limits the definition of birds to only animals, yet it still covers all examples including the negative ones. Third, the hypothesis is further restricted by adding $hasWing(X,Y)$ which limits birds to only animals that have wings; it still covers some negatives (e.g. a bat), though. Finally, by adding

$feather(Y)$, the hypothesis is restricted to animals that have wings covered in feathers; this restriction makes the hypothesis complete and consistent, and the learning is therefore terminated. An example of an ILP learner that uses the top-down strategy is the first-order inductive learner (FOIL) (Quinlan, 1990).

Bottom-up strategy

The bottom-up strategy uses specific-to-general refinement operators. This strategy can be demonstrated by using inverse resolution on the (simplified) bird example, as seen in Figure 2.3.

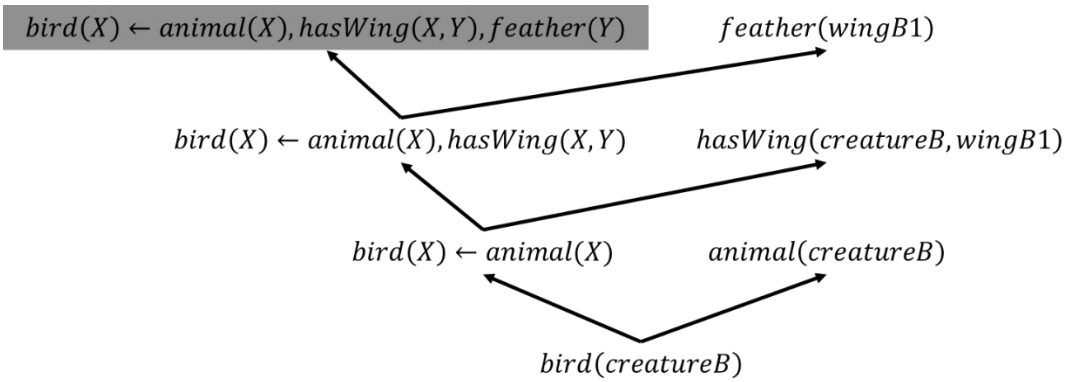


Figure 2.3: Learning the description of a bird using the bottom-up strategy.

In Figure 2.3, first, the positive example $creatureB$ is picked (typically at random). Second, the finding in B that $creatureB$ is an animal makes the hypothesis dictate that a bird is an animal. Third, $creatureB$ seem to have wings, which makes the hypothesis dictate that a bird is an animal that has wings. Finally, these wings are found to be covered in feathers, which makes the hypothesis dictate that a bird is an animal that has wings covered in feathers. An example of an ILP learner that uses this bottom-up approach is Golem (Muggleton and Feng, 1990).

Hybrid strategy

This approach is a mix of the top-down and bottom-up approaches, aiming to achieve the best of both approaches and to minimize their limitations. Typical examples of the hybrid approach are Progol (Muggleton, 1995) and Aleph (Srinivasan, 2007). In this approach, mode declarations are used to specify what predicates are and how these predicates can be used to build hypotheses. Mode declarations are used to construct the bottom clause, which reduces the hypothesis space and helps conduct the (top-down) hypothesis search more intelligently. In the bird example, the following mode declaration can be used "$:-modeb(2, hasWing(+animal, -wing))$?", which means that the predicate $hasWing$ only relates an animal to at most two wings; in other words, an animal can have at most two wings, unlike insects for example. In this mode declaration, an animal is an input variable based on "$+$" and a wing is an output variable based on "$-$". An input variable comes from another predicate. For the output variable, it is an input for another predicate. The differences between an input and an output variable can be seen in the following example:

$$bird(X) \leftarrow hasWing(X,Y), feather(Y)$$

Here, the variable X is an input variable for $hasWing$ since it comes from $bird(X)$, and the variable Y is an output variable for $hasWing$ since it is an input for $feather(Y)$. Regarding mode declarations, there can be one or more in a learning problem.

Other strategies

There are other approaches for ILP, such as the use of the genetic algorithm to learn hypotheses (Nezhad and Muggleton, 2002). In addition, a simple naïve (brute force) approach can be used, although it is discouraged because of the complexity of ILP learning.

A summary of ILP learning strategies can be seen in Table 2.1, and a classification of ILP learning strategies can be seen in Figure 2.4.

Table 2.1: A comparison of ILP learning strategies.

Learning Approach	Advantages	Limitations
Top-Down	• Simple to implement (generate and evaluate hypotheses)	• Possible generation and evaluation of large number of candidate hypotheses
Bottom-Up	• Avoid navigating large search spaces • Build the hypothesis by generalizing encountered examples	• Determining which a learning example is picked to generalize
Hybrid (combines Top-Down and Bottom-Up approaches)	• Efficient search (by restricting the hypotheses space)	• Reduced performance for huge number of examples – scalability issue • The configuration of mode declarations

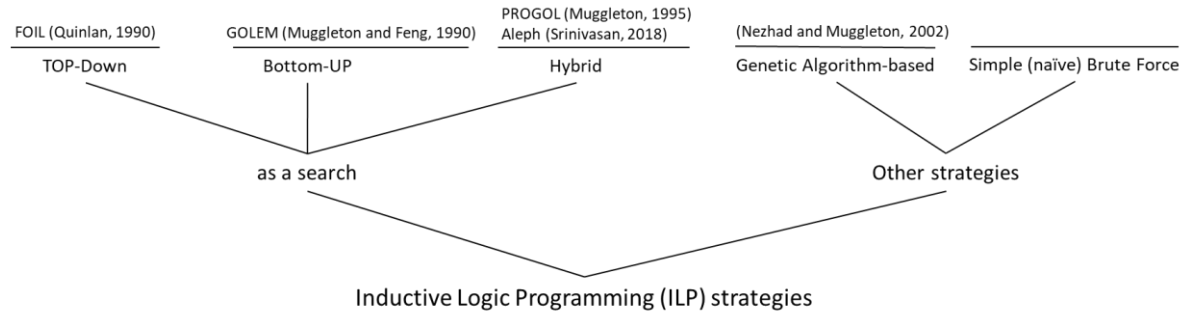


Figure 2.4: An overview of ILP learning strategies.

Since Progol employ measures for pruning areas of the search space using knowledge about the learning problem, which definitely improves ILP learning performance, a similar approach was used in this

thesis – namely, employing a top-down search on a restricted hypothesis space based on some knowledge about the learning problem.

2.1.3 ILP logic formalisms

As a logic-based ML technique, ILP can use virtually any logic formalism for knowledge and hypothesis representation. Many classical ILP learners use horn clauses as the logic formalism. The common ILP logic formalisms and their relation can be seen in Figure 2.5.

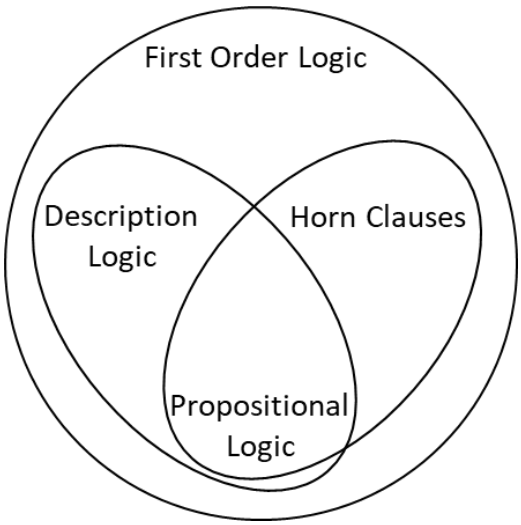


Figure 2.5: Common ILP logic formalisms.

First-order logic (FOL) is a very expressive logic that is able to represent complex concepts and structures using predicates and functions. Because of its complexity (in terms of inference), many ILPs are limited to their horn clauses subset, which excludes FOL functions. Horn clauses, on top of which Prolog (a commonly known logic programming system) is built, have been used in some ILP learners, such as Aleph. Prolog uses predicates of various lengths (number of terms) to represent knowledge. For example, $predicate(X)$ is a predicate with a single term (unary predicate), while $predicate(X, Y)$ is a predicate with two terms (binary predicate). A predicate can have many terms, and a term can be a variable (e.g. *X*, *Y*) or constant (e.g. David, Matthew). Predicates with only constant terms are known as ground facts such as $friendOf(jon, liam)$ which means Jon is a friend of Liam. The unary predicate $doctor(liam)$ means Liam is a doctor. A knowledge base in Prolog is known as a logic program. A logic program consists of ground facts and logic rules; both are known as "clauses". Ground facts are logic-based data assertions to the knowledge base. Logic rules can be used for many purposes, such as inferring (deducing) more ground facts from the knowledge base. Logic rules can also be used to ensure the consistency of the knowledge base. An example of a logic rule is

$father(X) :- male(X), hasChild(X, Y)$ which means a male who has a child is a father.

Description logic (DL) is a family of logic formalisms that are limited to unary and binary predicates in FOL (e.g. $man(X)$ and $friendOf(X, Y)$). In terms of expressivity, DL lies in the midpoint between

FOL and propositional logic. With this in mind, DL has the sufficient expressive power to represent many complex knowledge bases with less computational demands than FOL. More details about DL can be seen in section 2.2.

Propositional logic (PL) is also known as zero-order logic. It is limited to unary predicates in FOL (e.g. $smart$). Propositional logic has the advantage of being computationally inexpensive; however, it has very limited expressive power.

Many classical ILP learners use first order logic (particularly horn clauses) as the logic formalism to learn complex relations, such as first-order induction of decision lists (FOIDL) (Mooney and Califf, 1995), which learns first-order decision lists. ILP can be used with many types (or variations) of logic for a variety of learning applications. There are probabilistic ILPs that combine probabilities with logic (De Raedt *et al*., 2015). Some ILP learners work on the fuzzy logic (Serrurier *et al.*, 2007). ILP algorithms can also be used in description logic, such as DL-FOIL (Fanizzi, d'Amato and Esposito, 2008), which uses a modified version of a FOIL algorithm for DL learning. Another ILP learner in DL is the DL-Learner (Lehmann, 2010), which has several learning algorithms specifically designed for DL learning. Some DL-based learners are specifically designed for recommender system applications (Qomariyah and Kazakov, 2017), which use an ILP learning strategy inspired by Aleph/Progol learning.

Other simple logic learners (propositional learners) can also be used, such as candidate elimination algorithm (Mitchell, 1997, p. 32) and ID3 (Quinlan, 1986). See Table 2.2 for a comparison of the three common ILP logic formalisms. The meaning of the DL terms (e.g. TBox, RBox, ABox, axioms, and assertions) is discussed in section 2.2. An example use of the three logic formalisms to express the learning of the definition for a bird can be seen in Table 2.3.

Table 2.2: A comparison among common ILP logic formalisms.

	Propositional logic (PL)	**Description logic (DL)**	**First order logic (FOL)**
FOL predicate mapping	Unary predicates in FOL	Unary and binary predicates in FOL	N-ary predicates
Background knowledge	A single table containing unary relations (as columns) – attribute-value	Knowledge base (or OWL ontology, discussed in section 2.2)	Logic program (e.g. Prolog program)
Background knowledge schema	The table columns	TBox (and RBox) axioms	Predicates and logic rules

Background knowledge data	Table rows	ABox assertions	Ground facts
Learning examples	List of table rows	List of individuals	List of ground facts (in the target predicate)
Expressive power and reasoning complexity	Low	Medium	High
Example ILP learners	Candidate elimination algorithm and ID3	DL-FOIL and DL-Learner	FOIL, GOLEM, PROGOL, and ALEPH
Model representation	Propositional logic rule or tree model (for ID3)	Concept/class definition	Logic statement (definitive clause)

Table 2.3: Learning the definition of a bird in the three ILP logic formalism.

<table>
<tr><th>Logic Formalism</th><th>Propositional logic (PL)</th><th>Description logic (DL)</th><th>First-order logic (FOL)</th></tr>
<tr><td>Knowledge base</td><td>Single table containing features about all animals (including birds)</td><td>Knowledge base (OWL ontology) about all animals</td><td>Logic program containing ground facts and rules about all animals</td></tr>
<tr><td>Learning examples</td><td>For ‘isBird’, yes = positive example, no = negative example:
<table>
<tr><th>hasWing</th><th>hasBeak</th><th>hasFeathers</th><th>isBird</th></tr>
<tr><td>yes</td><td>yes</td><td>yes</td><td>yes</td></tr>
<tr><td>yes</td><td>yes</td><td>yes</td><td>yes</td></tr>
<tr><td>yes</td><td>no</td><td>no</td><td>no</td></tr>
<tr><td>no</td><td>yes</td><td>no</td><td>no</td></tr>
</table></td><td>Positives:
CreatureA: Animal
CreatureB: Animal
Negatives:
CreatureC: Animal
CreatureD: Animal
CreatureE: Animal</td><td>Positives:
bird(creatureA)
bird(creatureB)
Negatives:
bird(creatureC)
bird(creatureD)
bird(creatureE)</td></tr>
</table>

	no	no	no	no		
Hypothesis language	Conjunctions, disjunctions, and negations of the attributes: hasWing, hasBeak and hasFeathers.	**Concepts:** Animal Feather **Roles:** hasWing	**Predicates:** bird/1 feather/1 hasWing/2
PL rule output	$hasWings \wedge hasBeak \wedge hasFeathers$		
DL rule output	$Bird \equiv Animal \sqcap \exists hasWing(Feather)$		
FOL rule output	$bird(X) \leftarrow animal(X), hasWing(X,Y), feather(Y)$		

In Table 2.3, the PL rule output means that a bird is something that has wings, a beak, and feathers. In the DL rule, a bird is an animal that has a wing covered with feathers.

DL describes a bird in an identical way to horn clauses, only using different representation. This descriptive power of DL means that it is able to describe a complex concept with expressing power similar but not identical to that used by FOL. A key advantage in DL, though, lies in its reduced need for computing resources compared to horn clauses. As a result, it is the logic formalism that was used throughout this thesis.

2.1.4 Extending ILP logic formalisms

In this section, we briefly review extensions of ILP logic formalisms beyond conventional (definite clauses) approaches.

For logic-based extensions, ILP can be combined with other logic formalisms such as DL and PL, as discussed before, although, there are other logic-based approaches. ILP can be combined with Answer Set Programming (ASP) to learn very expressive hypotheses by taking advantage of ASP's expressive knowledge representation and efficient reasoning (Law, Russo and Broda, 2019). ASP is a type of

declarative programming where an ASP program is essentially a logic program (similar to Prolog's). There are ILP learners that combine ASP with probabilistic logic to learn probabilistic logic programs such as Credal-FOIL (Tuckey, Broda and Russo, 2020). Researchers in (Law, Russo and Broda, 2018a) have investigated the complexity and generality of the state-of-the-art for ASP-based ILP learning frameworks. The researchers have determined based on detailed analysis of the trade-off (between complexity and generality) for these frameworks, that a particular ASP learning framework with a certain trade-off is a recommended (over other frameworks) for a particular ASP learning problem (or scenario). Beside ASP-based ILP, ILP as a field, was also used for logic-based simulations (Alonso and Kudenko, 2000).

For ILP extensions with other machine learning approaches, ILP can be combined with other machine learning algorithms, such as reinforcement learning (RL), for a variety of applications; this combination is known as relational reinforcement learning (Džeroski, de Raedt and Driessens, 2001). ILP can also be combined with artificial neural networks (ANN) (DiMaio and Shavlik, 2004).

A classification of some of ILP logic formalisms and the possible extensions including other ML techniques can be seen in Figure 2.6.

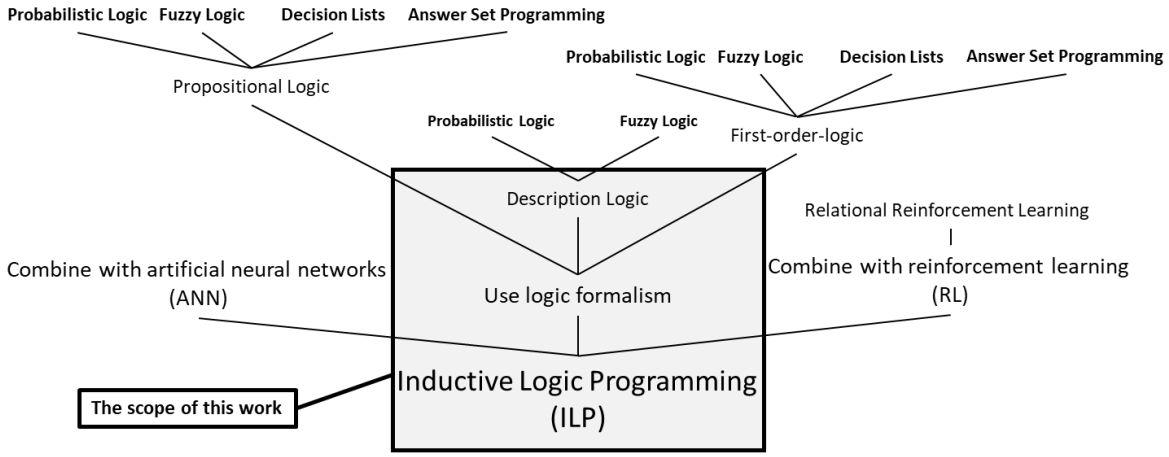


Figure 2.6: A classification of common ILP logic formalisms and their extension.

In Figure 2.6, each of the three logic formalisms can be extended to handle uncertainty using probabilistic logic and fuzzy logic to handle vagueness. On the other hand, a decision list can be roughly built on top of a logic formalism to describe a model as an ordered set of if-then rules.

2.1.5 Optimizing ILP learning

Many ILP learners suffer from the issue of a large search space. As a result, many ILPs apply different approaches to reduce the search space – for example, by setting an upper limit for the length of a hypothesis. In addition, ILP learners that use mode declarations (e.g. Aleph and Progol) can apply further reductions to the search space by limiting the predicates that can be used to construct hypotheses. Moreover, learners that use top-down searches may ignore a generated child hypothesis that is worse

than its parent hypothesis (e.g. based on the covered positive examples), consequently pruning a considerable part of the search space. Furthermore, enforcing an order of operands in hypotheses' conjunctions and disjunctions further eliminates the area of the search space by reducing the possible combinations of operands of the same hypothesis to a single combination. In addition, information about the background knowledge can be used to guide the search more intelligently – for example, by avoiding generating a conjunction of two disjoint operands. Moreover, optimizing score functions using domain-specific knowledge can help ILP scalability such as in the scalable ASP-based ILP learner FastLAS (Law *et al.*, 2020). In many real-world applications, finding noise-free data for ILP learning is difficult, therefore many approaches were developed to handle noisy data in ILP, such as (Law, Russo and Broda, 2018b) for ASP-based learning from noisy examples.

Although ILP learning has proved its success as an ML technique in many real-world applications, there are remaining challenges, especially regarding the issue of scalability.

A summary of ILP advantages and disadvantages can be seen in Table 2.4. Parallel computing, which is discussed in section 2.3, offers many opportunities with which the scalability challenge can be addressed.

Table 2.4: A summary of ILP learning advantages and challenges.

Advantages	**Challenges**
• White box model • Able to learn complex relations (e.g. chemical structures) • Can learn directly from relational databases	• Can be computationally expensive • Scalability issues • The complexity of engineering knowledge bases.

This thesis addresses the scalability issue for ILP learning by accelerating both hypothesis search and evaluation through the capabilities of parallel computing.

2.2 Description Logic

In this section, we provide necessary background information about DL. Because DL is the logic formalism used by this project to represent knowledge and models, a review of DL and its related concepts is therefore necessary.

Description logic is family of logic-based knowledge representation languages. In relation to other logic formalisms, DL is a subset of FOL, and its expressing power lies between propositional logic and FOL. A key advantage of DL is that it has acceptable computational complexity while also providing the

expressive power for the representation of many complex knowledge bases. Description logic is a widely used logic-based representation in semantic web technologies, particularly those associated with OWL (discussed in later sections).

2.2.1 Knowledge representation in DL

In DL, knowledge is represented as a knowledge base $\mathcal{KB}$ through a set of constructors – that is, concepts and roles (and sometimes concrete roles). These constructs can be combined to describe complex relations. Concepts describe unary relations, which are analogous to unary predicates in FOL. Roles represents binary relations, which are analogues to binary predicates in FOL. Roles provide the relation between instances, where instances are analogous to constants in FOL. In some DL languages, an additional DL construct can be used (i.e. concrete roles). Similar to roles, concrete roles represent binary relations; however, these relations only relate an instance to a value (e.g. number, date, and text). See Table 2.5 for the mapping between DL notation and its FOL equivalence.

Table 2.5: A mapping of DL terminology to FOL equivalents.

DL Construct		**DL Example**	**FOL (Prolog) example**
Concept	Unary predicate	*Adam: Male* *Eve: Female* *John: Engineer*	*male(adam)* *female(eve)* *engineer(john)*
Role	Binary predicate	*(David, John): friendOf* *(David, Jon): hasSon* *(Craig, Tom): parentOf*	*friendOf(david,john)* *hasSon(david,jon)* *parentOf(craig,tom)*
Concrete role		*(David, 100.4): hasWeight* *(Jon, 1): isTall* *(Craig, 1984): bornIn*	*hasWeight(david,100.4)* *isTall(jon,1)* *bornIn(craig, 1984)*

DL employs the notion of an interpretation $\mathcal{I} = (\triangle^{\mathcal{I}}, .^{\mathcal{I}})$ to model knowledge, where $\triangle^{\mathcal{I}}$ is known as the interpretation domain, which is a non-empty set of instances. $.^{\mathcal{I}}$ is known as the interpretation function that maps 1) every instance *a* such that $a^{\mathcal{I}} \in \triangle^{\mathcal{I}}$; 2) every concept A to a set of instances, such that $A^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$; and 3) every role R to a binary relation such that $R^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$. An interpretation can be visualised as a directed graph. An example of an interpretation and its visualization can be seen in Figure 2.7.

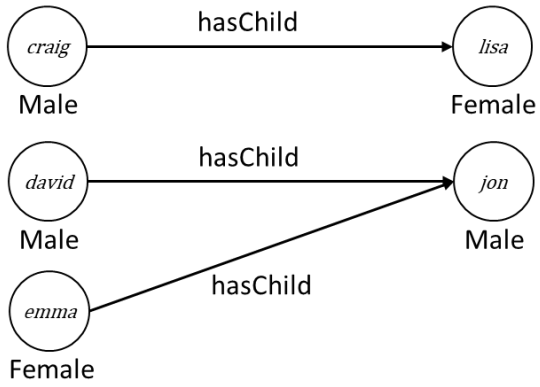
$\triangle^{\mathcal{I}} = \{david, emma, jon, craig, lisa\}$,

$Male^{\mathcal{I}} = \{david, jon, craig\}$,

$Female^{\mathcal{I}} = \{emma, lisa\}$,

$hasChild^{\mathcal{I}} = \{(david, jon), (emma, jon), (craig, lisa)\}$.

An example interpretation $\mathcal{I}$



Interpretation $\mathcal{I}$ graph representation

Figure 2.7: An example of an interpretation and its visualization.

An interpretation $\mathcal{I}$ can be modified in terms of its $\triangle^{\mathcal{I}}$ and/or $.^{\mathcal{I}}$ to obtain other interpretations. There are many variations of DL languages, each of which has its own expressive power; more expressive DLs (typically) subsume less expressive ones. What determines a DL language's expressivity is the available DL constructors to represent a given knowledge.

2.2.2 DL languages

Many popular DL languages build upon or extend languages, such as $\mathcal{AL}$ (attributive language) and $\mathcal{EL}$ (existential language). A well-known DL variation is $\mathcal{ALC}$ (an extension of the $\mathcal{AL}$ language), when $\mathcal{ALC}$ is combined with transitive roles, the $\mathcal{S}$ language is achieved. See Table 2.6 for a list of some DL constructors, where A is any atomic (simple) concept, C and D are any simple or complex concepts, r and s are roles, n is any integer number > 0, and a is an instance. DL constructors are the building blocks used to describe relations in DL.

Table 2.6: Some DL constructers and their naming (Baader *et al.*, 2017, p. 232).

Name	**Syntax**	**Symbol**	**Basic DL languages support**			**Semantics**
			$\mathcal{AL}$	$\mathcal{EL}$	$\mathcal{S}$	
Top concept	$\top$		Yes	Yes	Yes	$\triangle^{\mathcal{I}}$
Bottom concept	$\bot$		Yes		Yes	$\emptyset$
Conjunction	$C \sqcap D$		Yes	Yes	Yes	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Atomic negation	$\neg A$		Yes		Yes	$\triangle^{\mathcal{I}} \setminus A^{\mathcal{I}}$
Value restriction	$\forall r.C$		Yes		Yes	$\{d \in \triangle^{\mathcal{I}} \mid \forall e \in \triangle^{\mathcal{I}}.(d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$
Disjunction	$C \sqcup D$	$\mathcal{U}$			Yes	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Negation	$\neg C$	$\mathcal{C}$			Yes	$\triangle^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Existential restriction	$\exists r.C$	$\mathcal{E}$		Yes	Yes	$\{d \in \triangle^{\mathcal{I}} \mid \exists e \in \triangle^{\mathcal{I}}.(d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$
Unqualified number restriction	$\leq n\, r$ $\geq n\, r$	$\mathcal{N}$				$\{d \in \triangle^{\mathcal{I}} \mid \#\{e \mid (d,e) \in r^{\mathcal{I}}\} \leq n\}$ $\{d \in \triangle^{\mathcal{I}} \mid \#\{e \mid (d,e) \in r^{\mathcal{I}}\} \geq n\}$
Qualified number restriction	$\leq n\, r.C$ $\geq n\, r.C$	$\mathcal{Q}$				$\{d \in \triangle^{\mathcal{I}} \mid \#\{e \mid (d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\} \leq n\}$ $\{d \in \triangle^{\mathcal{I}} \mid \#\{e \mid (d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\} \geq n\}$
Nominal	$\{a\}$	$\mathcal{O}$				$\{a^{\mathcal{I}}\}$
Inverse role	r^{-}	$\mathcal{I}$				$\{(e,d) \in \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}} \mid (d,e) \in r^{\mathcal{I}}\}$
Role inclusion	$r \sqsubseteq s$	$\mathcal{H}$				$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
Complex role inclusion	$r_1 \circ \ldots \circ r_n \sqsubseteq s$	$\mathcal{R}$				$r_1^{\mathcal{I}} \circ \ldots \circ r_n^{\mathcal{I}} \subseteq s^{\mathcal{I}}$

Functionality	$\mathrm{Func}(r)$	$\mathcal{F}$				$(d,e) \in r^{\mathcal{I}} \wedge (d,f) \in r^{\mathcal{I}} \Longrightarrow e = f$
Transitivity	$\mathrm{Trans}(r)$	R^{+}			Yes	$(d,e) \in r^{\mathcal{I}} \wedge (e,f) \in r^{\mathcal{I}} \Longrightarrow (d,f) \in r^{\mathcal{I}}$

In DL, every instance in the knowledge base (e.g. Craig, David and Emma) is a member of the top concept ($\top$), whereas the bottom concept ($\bot$) has no members (i.e. an empty set, $\emptyset$). The difference between atomic negation $\neg A$ and full negation $\neg C$, is that atomic negation is a restricted form of negation limited to only negating simple concepts (e.g. a doctor, a teacher, etc.), while full negation can negate a simple concept or a complex concept (e.g. a complete DL hypothesis constructed using multiple constructors). In terms of number restrictions, unqualified number restriction $\leq n\, r$ is a restricted form of qualified number restriction limited to the top concept only, i.e. $\leq n\, r$ is the same as $\leq n\, r.\top$. Qualified number restriction $\leq n\, r.C$ can use any concept (the top concept, atomic concept or a full DL hypothesis) as a C.

Nominals $\{a\}$ refer to an instance a in the knowledge base. Nominals can be used in role restrictions e.g. $\exists r.\{a\}$ or simply $\exists r.a$ such that $\{d \in \triangle^{\mathcal{I}} \mid \exists e \in \triangle^{\mathcal{I}}.(d,a) \in r^{\mathcal{I}} \wedge a \in \triangle^{\mathcal{I}}\}$. More nominals can be used in a restriction e.g. $\exists friendOf.\{David, Jacob\}$ which will return all instances that have at least David and Jacob as friends. For inverse role r^{-}, the inverse of a role is defined as $(r^{-})^{\mathcal{I}} = \{(y,x) \mid (x,y) \in r^{\mathcal{I}}\}$. Inverse roles can be used in role restrictions such as $\exists r^{-}.C$ for existential restriction. For example, $\exists hasChild.Male$ will return all parents (instances) that have a male child, whereas $\exists hasChild^{-}.Male$ will return all male children related to a parent through $hasChild$ role. In other words, $\exists hasChild.Male$ returns the parents of male children, while $\exists hasChild^{-}.Male$ returns their male children.

In terms of role inclusions, $r \sqsubseteq s$ describe role hierarchy ($r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$). For example, using the role inclusions $hasFather \sqsubseteq hasParent$ and $hasMother \sqsubseteq hasParent$, means that $hasFather$ and $hasMother$ are types of $hasParent$. In other words, $hasParent$ subsumes all instances of $hasFather$ and $hasMother$. In a complex role inclusion ($r_1 \circ \ldots \circ r_n \sqsubseteq s$), multiple roles can be used such that $R^{\mathcal{I}} \circ S^{\mathcal{I}} = \{(a,c) \mid$ there is some b with $(a,b) \in R^{\mathcal{I}}$ and $(b,c) \in S^{\mathcal{I}}\}$ in the role inclusion such as $R \circ S \sqsubseteq s$. For example, $SisterOf \circ MotherOf \sqsubseteq auntOf$ means a sister of someone's mother is an aunt of that someone. It is worth noting that the constructors in Table 2.6 from role inclusion $r \sqsubseteq s$ until Transitivity $\mathrm{Trans}(r)$, are concerned with constraining a DL knowledge base which will affect the type of DL hypotheses allowed to be constructed; these DL constructors and other remaining ones in Table 2.6 are discussed in the next sections.

A DL language is described as a combination of the symbols (constructors) in Table 2.6. For example, the DL language $\mathcal{ALC}$ is able to use $\mathcal{AL}$'s constructors in addition to applying the negation constructor on complex concepts ($\mathcal{C}$); in a similar vein, the language $\mathcal{ALCQ}$ is able to express a qualified cardinality restriction ($\mathcal{Q}$) on roles (e.g. $\geq 3\ hasChild.Person$, which obtains all parents who have at least three children). Moreover, when $\mathcal{ALC}$ is extended to support transitive roles (R^+), it is called $\mathcal{S}$. In this project, the $\mathcal{ALCQ}^{(\mathcal{D})}$ language was used to represent knowledge and models. It is an extension of the $\mathcal{ALCQ}$ language, and it supports concrete roles through the superscript ($\mathcal{D}$) – for example, David was born in 1986, or in DL syntax *(David, 1986): bornIn*. The selected DL language is able to sufficiently express many real-world concepts while still retaining an acceptable computational complexity. In addition, when it is extended to support role hierarchy through role inclusion ($\mathcal{H}$), it becomes the representation language used by the state of the art for DL inductive learning – that is, the DL-Learner (see chapter 3 for more detail).

DL has many constructors, some of which are not shown in the table above. In this project, we are only concerned with reviewing the relevant constructors. A general rule in DL dictates that adding more expressive constructors to a language will increase its computational complexity.

2.2.3 DL constructors

The DL constructors can be applied to a set of concepts and roles to describe complex relations (DL hypotheses). See Table 2.7 for examples of DL hypotheses constructed using DL operators.

Table 2.7: Examples of DL hypotheses.

DL Constructor Type	**DL constructor (operator)**	**Example**
Concept	Conjunction	$Man \sqcap Teacher$
	Disjunction	$Man \sqcup Woman$
	Negation	$\neg Teacher$
Role	Existential restriction	$\exists hasChild.Female$
	Value restriction	$\forall hasChild.Engineer$
	MIN Number restriction	$\geq 3\ friendOf.Person$
	MAX Number restriction	$\leq 3\ friendOf.Person$

In Table 2.7, the conjunction ($Man \sqcap Teacher$) represents the instances that are members of both the *Man* and the *Teacher* concepts. For the disjunction ($Man \sqcup Woman$), the members are the instances

of either the *Man* or the *Woman* concept. For the negation ($\neg Teacher$), the members are all instances that do not belong to the *Teacher* concept.

In the existential restriction ($\exists hasChild.Female$), the members are all instances that have a female child. In the value restriction ($\forall hasChild.Engineer$), all the instances that have only *Engineer* children are the members. This also includes instances that have no children at all because of the open-world assumption (discussed below). For the number restriction ($\geq 3\ friendOf.Person$), the members are all instances that have at least three friends. A role restriction can be written using two acceptable syntaxes $r.C$ or $r(C)$, such as $\exists r.C$ or $\exists r(C)$ for existential restriction, $\forall r.C$ or $\forall r(C)$ for value restriction, and $\leq n\ r.C$ or $\leq n\ r(C)$ for a MIN number restriction (the same applies for other number restrictions).

A DL hypothesis is essentially a (complex) concept (such as the examples in Table 2.7). A hypothesis is constructed using one or multiple DL constructors on (simple or complex) concepts. The simplest form of a hypothesis is being an atomic concept (e.g. $Teacher$, $Engineer$, etc.). A hypothesis can be of any arbitrary length *n*, e.g. $C_1 \sqcap ... \sqcap C_n$ or $C_1 \sqcup ... \sqcup C_n$ where C is any simple concept in the knowledge base or a complex concept (such as the examples in Table 2.7). In addition, a hypothesis can be of any arbitrary depth, where (simple or complex) concepts can be used in a nested way to describe more complex concepts, e.g. $C \sqcap \exists r.E$, $C \sqcup \exists r.E$, $\exists r(E \sqcap C)$, $\leq v\ r(E \sqcap C)$, etc; where E is any concept, r is any role, and v is any value for a number restriction. Concepts can be nested more such as $C \sqcap \exists r.(\leq v\ s(E \sqcap D))$ which can be nested more into $((C \sqcap \exists r.(\leq v\ s(E \sqcap D))) \sqcup B) \sqcap \neg F$. For an example of a nested hypothesis, $\exists hasChild(\exists hasChild(Person))$ describes a grandparent, which can be restricted to grandfather by adding a conjunction with a $Male$ concepts $Male \sqcap \exists hasChild(\exists hasChild(Person))$, to become a grandfather (a male grandparent). Concepts can be nested to the grandfather hypothesis, e.g. $Male \sqcap \exists hasChild(\exists hasChild((Teacher \sqcup engineer) \sqcap \forall friendOf.EducatedPerson))$ which means a grandfather of a teacher or an engineer whose all of their friends are educated.

There is no particular limit for a hypothesis depth or length. Although, in practice, a hypothesis may be limited to a particular length or depth to improve readability and to reduce computational complexity (related to DL reasoning, see section 2.2.4 for more details).

In DL, the hypothesis can exist as a concept definition. For example, the concept definition (DL hypothesis):

$$TeacherWithFamily \equiv Teacher \sqcap \exists marriedTo(Person) \sqcap = 1\ hasChild.Male \sqcap = 1\ hasChild.Female$$

which means that the definition of the complex concept $TeacherWithFamily$ is someone who is a teacher and married to a person and has exactly one son and one daughter.

Terminological box and the assertional box

In DL, the knowledge representation is divided into terminological box (TBox), relational box (RBox), andassertional box (ABox). The TBox is a set of concept axioms that constrain the concepts of a knowledge base. The RBox is used to constrain the roles of a knowledge base. Both TBox and RBox provide the constraints or rules governing the knowledge base. ABox holds the knowledge bases' assertions (i.e. data entries). For example, consider the following TBox concept definition axiom:

$$smallFamily \equiv Teacher \sqcap \exists marriedTo(Person) \sqcap = 1\ hasChild.Male \sqcap = 1\ hasChild.Female$$

the ABox will contain all the instances of the *smallFamily* concept (i.e. the ones that satisfy its constraints). See Table 2.8 for an overview of the TBox, RBox, and ABox. The ABox is explained first since it lays the foundation for the discussion of the TBox and RBox.

Table 2.8: An overview of TBox, RBox and ABox (Baader *et al.*, 2017, p. 231).

	Name	**Syntax**	**Semantic**
TBox	General concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
	Concept definition	$A \equiv C$	$A^{\mathcal{I}} = C^{\mathcal{I}}$
RBox	Role inclusion	$r \sqsubseteq s$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
	Role disjointedness	*Disj(r,s)*	$r^{\mathcal{I}} \cap s^{\mathcal{I}} = \emptyset$
	Role transitivity	*Trans(r)*	$(d,e) \in r^{\mathcal{I}} \wedge (e,f) \in r^{\mathcal{I}} \Longrightarrow (d,f) \in r^{\mathcal{I}}$
	Role functionality	*Func(r)*	$(d,e) \in r^{\mathcal{I}} \wedge (d,f) \in r^{\mathcal{I}} \Longrightarrow e = f$
	Role reflexivity	*Ref(r)*	$d \in \triangle^{\mathcal{I}} \Longrightarrow (d,d) \in r^{\mathcal{I}}$
	Role irreflexively	*Irref(r)*	$d \in \triangle^{\mathcal{I}} \Longrightarrow (d,d) \notin r^{\mathcal{I}}$
	Role symmetry	*Sym(r)*	$(d,e) \in r^{\mathcal{I}} \Longrightarrow (e,d) \in r^{\mathcal{I}}$
	Role antisymmetry	*Asym(r)*	$(d,e) \in r^{\mathcal{I}} \Longrightarrow (e,d) \notin r^{\mathcal{I}}$
ABox	Concept assertion	$a{:}\,C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
	Role assertion	$(a,b){:}\,r$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$

In Table 2.8, the ABox is used to add assertions for concepts or roles into the knowledge base. An example of a concept assertion is $David{:}\,Person$, which means the instance *David* is a member of the

Person concept. For role assertions, an example would be $(david, john)$: $hasChild$, which means *David* has a child, *John*; the subject and the object of this role assertion are David and John, respectively.

In terms of TBox, the concept axioms are used to define concepts and their constraints, such as the concept definition $Engineer \equiv Person \sqcap \exists hasDegree(Engineering)$, meaning an engineer is a person who has a degree in engineering. For the concept inclusion axiom, an example would be $PhDStudent \sqsubseteq Student \sqcap \exists doResearchIn(\top)$, which means that a PhD student can be a student who does research in anything ($\top$, the top concept).

In terms of role axioms in the RBox, an example of role inclusion is $hasBrother \sqsubseteq hasSibling$, which means that the role *hasBrother* is a type (sub-role) of *hasSibling*; that is, the role assertions of $hasBrother$ are also considered the role assertions of $hasSibling$. In the role axiom disjointedness, given the two roles *r1* and *r2*, and the two instances *x* and *y*, the role assertions $(x, y): r1$ and $(x, y): r2$ are strictly not allowed. For example, the two roles *fatherOf* and *motherOf* are disjointed – that is, an instance cannot be a father and a mother of someone at the same time. For the role transitivity, a popular example is the *hasAncestor* role, as seen in Figure 2.8. An *individual (instance) A* is linked to an *individual B*, who is in turn linked to *individual C* through the same role. From this, it can be inferred that *individual A* is also indirectly linked to *individual C* though the same role.

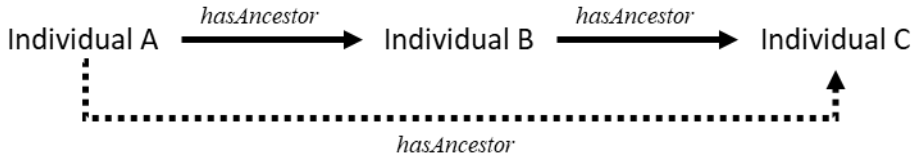


Figure 2.8: A popular example of a transitive role.

In functionality, the role links at most one individual to another. See Figure 2.9 for an example in which *Individual A* has one biological father *Individual B*.

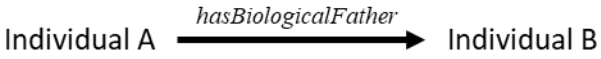


Figure 2.9: An example of a functional role.

For reflexivity, the role links an individual to its self. For example, an individual can be responsible for himself or herself, as shown in Figure 2.10.

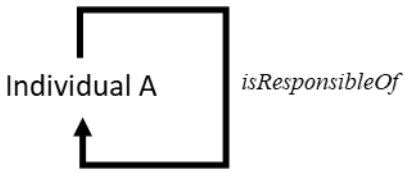


Figure 2.10: An example of a reflexive role.

Irreflexivity, the opposite of reflexivity, indicates that an individual is not allowed to link to itself. For example, a person cannot be a parent of himself or herself. In symmetric roles, the role goes in both directions. For example, *Individual A* is a friend of *Individual B*, which also implies that *Individual B* is a friend of *Individual A*. In an asymmetric role, the role can go only one direction. such as the *parentOf* role. In addition to role axioms, a role can be constrained in terms of its subject and object to a specific set of individuals by specifying the domain and range concepts, respectively. For example, the domain (role subject) for *hasPet* can constrained to only *Person* individuals, and the range (role object) to only *Animal* individuals, which means that the role *hasPet* can only relate a *Person* to an *Animal*.

In terms of concrete roles, since they are a type of a binary relation, they have similar constructors that of roles. For example, a concrete role assertion is $(a, v)\colon d$, which links an individual to a data value (literal). In terms of their constraints, they have similar constraints to that of roles in the RBox. Such constraints include disjointedness, functionality, hierarchy (inclusion), and domain and range constraints.

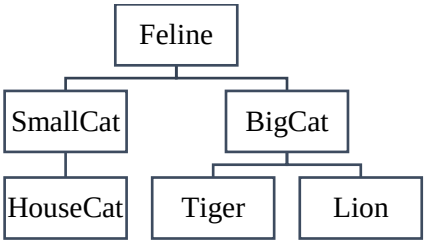
Since the TBox axioms constrain the concepts of a knowledge base and since RBox axioms constrain roles and concrete roles, a DL reasoner is used to check the knowledge base's ABox against its constraints in the TBox and RBox. In addition, a DL reasoner can use the TBox and RBox to infer new facts (assertions). In other words, a DL reasoner ensures the consistency of a knowledge base and infers new assertions.

2.2.4 DL reasoning

In DL, a reasoner can provide at least some of the following reasoning tasks: classification, satisfiability, instance retrieval (check), and realization. For classification, the concepts in TBox are used to compute concept hierarchy (or subsumption hierarchy). For example, in the following TBox:

$$TBox = \left\{ \begin{matrix} HouseCat \sqsubseteq SmallCat, \\ Tiger \sqsubseteq BigCat, \\ Lion \sqsubseteq BigCat, \\ SmallCat \sqsubseteq Feline, \\ BigCat \sqsubseteq Feline \end{matrix} \right\}$$

a DL reasoner will infer the following concept hierarchy:



Moreover, a DL reasoner will also infer that all individuals of *HouseCat* are also members of the *Feline* concept. For knowledge base satisfiability, the knowledge base is checked if it is consistent – that is,

that it has an interpretation $\mathcal{I}$ that entails the knowledge base $\mathcal{KB}$, such that $\mathcal{I} \vDash \mathcal{KB}$. In other words, a knowledge base is said to be consistent (satisfiable) if it does not contradict itself. For example, in the following knowledge base:

$\mathcal{KB}(\mathcal{T}, \mathcal{A})$	
$\mathcal{T} = \begin{Bmatrix} Female \sqsubseteq \neg Male, \\ Male \sqsubseteq \neg Female \end{Bmatrix}$	$\mathcal{A} = \begin{Bmatrix} Adam: Male, \\ Eve: Female, \end{Bmatrix}$

the axioms in the TBox mean that the two classes $Male$ and $Female$ are disjointed (i.e. a person cannot be a male and female at the same time). For the ABox, there are two assertions: Adam is a male and Eve is a female. This knowledge base is consistent, but adding the assertion $Adam: Female$ or $Eve: Male$ would make the knowledge base inconsistent and therefore unsatisfiable by violating its TBox constraints. In addition, a DL reasoner can check a concept's satisfiability; a concept is said to be satisfiable if it has a member individual in a consistent knowledge base. Regardless of the knowledge being represented, an individual that belongs to a concept C and its negation $\neg C$ will render the knowledge base inconsistent. For instance retrieval, the individuals that belong to a specific concept are retrieved. For example, consider the following ABox:

$$\mathcal{A} = \begin{Bmatrix} Adam: Male, \\ Cain: Male, \\ Eve: Female, \\ Emma: Female \end{Bmatrix}$$

When the reasoner is asked to retrieve all members of the $Male$ concept, it will return $\{Adam, Cain\}$. For the realization task, the reasoner will use the rules of the knowledge base to infer by deducing new assertions. For example, in the following knowledge base:

$\mathcal{KB}(\mathcal{T}, \mathcal{A})$	
$\mathcal{T} = \begin{Bmatrix} Female \sqsubseteq \neg Male, \\ Male \sqsubseteq \neg Female, \\ EducatedPerson \equiv \exists hasDegree. \top \end{Bmatrix}$	$\mathcal{A} = \begin{Bmatrix} Adam: Male, \\ Eve: Female, \\ (Adam, Biology): hasDegree \\ (Eve, Chemistry): hasDegree \end{Bmatrix}$

a reasoner will deduce based on the axiom $EducatedPerson \equiv \exists hasDegree. \top$ (which means that an educated person is someone who has a degree in something) that Adam and Eve are members of this concept since they both have a degree in something, therefore they are members of the $EducatedPerson$ concept.

Reasoning tasks that involve only concept axioms will use the TBox in their reasoning. For roles and concrete roles' axioms, the RBox is also used. The architecture of a DL system can be seen in Figure 2.11.

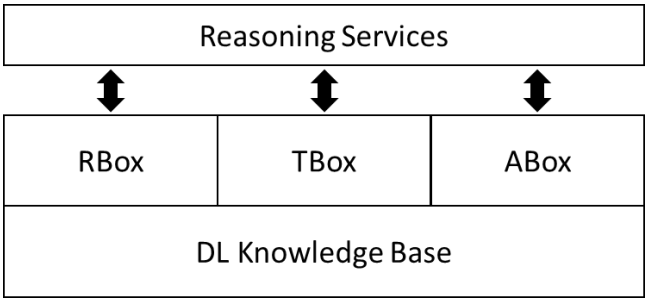


Figure 2.11: The architecture of a DL system.

Reasoners have been used in many real-world applications for many different purposes, such as to ensure the knowledge being captured is consistent. In addition, reasoners can act as an expert system to infer answers for many questions about a knowledge base. Moreover, DL reasoners are also used by DL-based ILP learners (e.g. the DL-Learner) to pre-process the DL knowledge base (to prepare for ILP learning). There are many DL reasoners with different reasoning algorithms. A list of common DL reasoners is provided in Table 2.9.

Table 2.9: A list of common DL reasoners.

<table>
<tr><th rowspan="2">DL-based OWL reasoner</th><th colspan="2">DL language</th><th rowspan="2">Reasoning algorithm</th></tr>
<tr><th>Language name</th><th>Language expressivity</th></tr>
<tr><td>Konclude (Steigmiller, Liebig and Glimm, 2014)</td><td>$\mathcal{SROIQV}$</td><td rowspan="4">Highest relative expressivity

↑

Lowest relative expressivity</td><td rowspan="4">Tableau Algorithms</td></tr>
<tr><td>HermiT (Glimm et al., 2014)</td><td rowspan="3">$\mathcal{SROIQ}^{(\mathcal{D})}$</td></tr>
<tr><td>Pallet (Sirin et al., 2007)</td></tr>
<tr><td>FACT++ (Tsarkov and Horrocks, 2006)</td></tr>
<tr><td>ELK (Kazakov, Krötzsch and Simancik, 2012)</td><td>$\mathcal{EL}$</td><td>Consequence-based</td></tr>
</table>

Currently, most expressive DL reasoners (e.g. Konclude and HermiT) are tableau-based algorithms, whereas many reasoners for $\mathcal{EL}$ are consequence-based. The tableau algorithm contains a set of one or more expansion rules for each DL constructor, where a single expansion rule is applied at a time on the knowledge base until no other expansion rule can be applied. The tableau algorithm will stop when no expansion rule can be applied or when an inconsistency is found – for example, an individual belongs to a concept and its negation (unsatisfiable). An example of expansion rules for checking the consistency of ABox $\mathcal{A}$ in the $\mathcal{ALC}$ language is shown in Figure 2.12.

$$\sqcap -rule\text{: } if\ 1.\, a\text{: } C \sqcap D \in \mathcal{A}, and\ 2.\, \{a\text{: } C, a\text{: } D\} \nsubseteq \mathcal{A}\ then$$

$$\mathcal{A} \longrightarrow \mathcal{A} \cup \{a\text{: } C, a\text{: } D\}$$

$$\sqcup -rule\text{: } if\ 1.\, a\text{: } C \sqcup D \in \mathcal{A}, and\ 2.\, \{a\text{: } C, a\text{: } D\} \cap \mathcal{A} = \emptyset\}\ then$$

$$\mathcal{A} \longrightarrow \mathcal{A} \cup \{a\text{: } X\}\ for\ some\ X \in \{C, D\}\}$$

$$\exists -rule\text{: } if\ 1.\, a\text{: } \exists r.\, C \in \mathcal{A}, and\ 2.\, there\ is\ no\ b\ such\ that\ \{(a, b)\text{: } r.\, b\text{: } C\} \subseteq \mathcal{A}\ then$$

$$\mathcal{A} \longrightarrow \mathcal{A} \cup \{(a, d)\text{: } r, d\text{: } C\}\ where\ d\ is\ new\ in\ \mathcal{A}$$

$$\forall -rule\text{: } if\ 1.\, \{a\text{: } \forall r.\, C, (a, b)\text{: } r\} \subseteq \mathcal{A}, and\ 2.\, b\text{: } C \notin \mathcal{A}\ then$$

$$\mathcal{A} \longrightarrow \mathcal{A} \cup \{b\text{: } C\}$$

Figure 2.12: The expansion rules for $\mathcal{ALC}$ ABox consistency (Baader *et al.*, 2017, p. 73).

In Figure 2.12, the four expansion rules are used to check the consistency of an $\mathcal{ALC}$ ABox where each one corresponds to a particular constructor (e.g. the $\sqcap -rule$ rule is for conjunction). The algorithm checks each expansion rule to see if its conditions are met – that is, if the ABox contains assertion (s) with particular characteristics, a specific expansion rule is triggered. An ABox assertion (s) is generated as a result of applying an expansion rule, where only a single rule is triggered at a time. For example, in the $\sqcap -rule$, the encountered assertion is checked if it is in the form $a\text{: } C \sqcap D$; if so, then ABox is checked to determine whether it contains the two following assertions $\{a\text{: } C, a\text{: } D\}$, if they do not exist, then they are added into the ABox as such: $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a\text{: } C, a\text{: } D\}$. After any rule application, the ABox is checked again for any matching rule, which means that the same expansion rule may be triggered multiple times (for different assertions). The algorithm terminates when no further rules can be applied or when a clash is found (i.e. an individual a and a concept C the ABox, contains the assertions $\{a\text{: } C, a\text{: } \neg C\}$). Once the algorithm terminates (regardless of clash existence), the knowledge base is said to be complete. A knowledge base that is complete and has no clashes is said to be consistent. The order of applying expansion rules has no effect on the algorithm's final outcome, although it may affect the reasoning performance. In order for the tableau algorithm to support more DL constructs, it has to

extend its expansion rules. In some DL reasoners, such as HermiT, the hypertableau reasoning is employed, which is an improvement of the tableau algorithm.

For consequence-based reasoning, there is a set of deduction rules that are applied to the knowledge base, which can be extended to support more expressive ($\mathcal{EL}$) languages. An example of a (classification) deduction rule is $\frac{A_1 \sqsubseteq A_2 \; A_2 \sqsubseteq A_3}{A_1 \sqsubseteq A_3}$, which means that if $A_1 \sqsubseteq A_2$ and $A_2 \sqsubseteq A_3$, then add $A_1 \sqsubseteq A_3$ into the TBox. For example, consider the following TBox:

$$\mathcal{T} = \begin{Bmatrix} Lion \sqsubseteq Feline, \\ Feline \sqsubseteq Carnivore \end{Bmatrix}$$

The algorithm will infer from the deduction rule above that since $Lion \sqsubseteq Feline$ and $Feline \sqsubseteq Carnivore$, $Lion \sqsubseteq Carnivore$ should be added into the TBox. Consequence-based reasoning is similar to the tableau algorithm, where the output of a rule is an input to other rules. Consequence-based reasoning is commonly used with the $\mathcal{EL}$ language and its variations. Tableau reasoning (including the hypertableau) is commonly used with $\mathcal{ALC}$ languages and its derivatives.

There are other DL reasoning techniques, such as automata (Glimm, Horrocks and Sattler, 2008) and resolution (the one used in FOL). However, since DL reasoning is a large topic, only a brief review is provided for the two most popular reasoning techniques (i.e. tableau and consequence-based). The more expressive the DL reasoning is (in terms of the DL language), the more computationally expensive it becomes; this effect is also amplified by the size of the knowledge base. Most DL reasoners are sequential in terms of their processing. In contrast, the only DL reasoner that employs parallel computing for reasoning in expressive DL languages is the Konclude reasoner, which can support DL reasoning in the $\mathcal{SROIQV}$ language. Many DL reasoners adopt deductive reasoning to infer new facts given a set of rules and the available data. In this project, we used inductive reasoning in the form of ILP learning. Moreover, we used deductive reasoning to prepare the knowledge base for ILP learning.

The use of description logic is commonly associated with semantic web technologies – OWL, in particular. In other words, the use of description logic in the real world is done through the OWL language, which is considered in the next section.

2.2.5 DL and the semantic web (OWL)

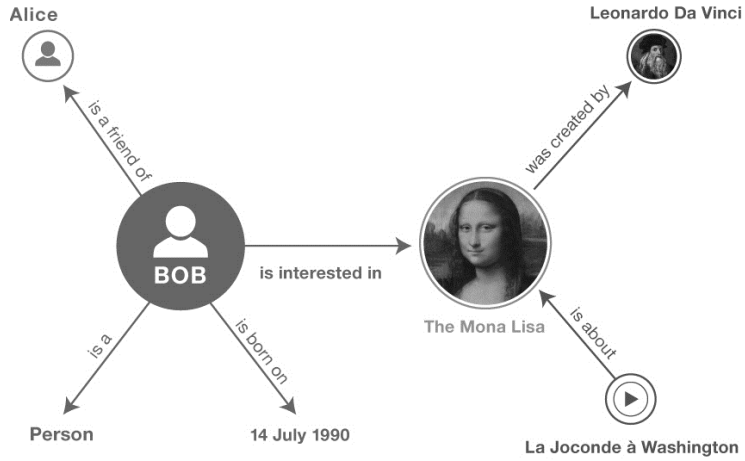
The web ontology language (or OWL)[4] is a family of languages that semantically represent knowledge as ontologies, and OWL is not bound to a specific data format. For example, OWL ontologies can be described using JSON, XML, LaTeX, and others. In other words, OWL is not bound to a specific file format. However, OWL typically employs the resource description framework (RDF)[5] format as its graph data model. In RDF, the data is typically represented through RDF triples, where a single triple

[4] https://www.w3.org/OWL/
[5] https://www.w3.org/RDF/

takes the form $subject, predicate, object$ which relate a subject to an object through a unary or a binary predicate. Each part of the triple is an RDF resource that has its own uniform resource identifier (URI). An RDF resource can be anything that has a URI, including text, numbers, pictures, videos, and so on. In order to demonstrate RDF triples, a simplified example for describing a person interested in the Mona Lisa painting can be seen in Figure 2.13, where the triples are represented in the form **<subject> <predicate> <object>**.

```
<Bob> <is a> <person>.

<Bob> <is a friend of> <Alice>.

<Bob> <is born on> <the 4th of July 1990>.

<Bob> <is interested in> <the Mona Lisa>.

<the Mona Lisa> <was created by> <Leonardo da Vinci>.

<the video 'La Joconde à Washington'> <is about> <the Mona Lisa>
```

RDF triples representation



RDF graph representation

Figure 2.13: RDF example for describing a person interested in the Mona Lisa painting.[6]

The query language SPARQL[7] provides various facilities to query the RDF data. Instead of using RDF to represent OWL ontologies, other data representations can be used as well (e.g. Manchester syntax,[8] OWL/XML, and functional-style syntax). With regard to the relation between DL and OWL, DL knowledge bases can be considered ontologies in OWL, where concepts and roles in DL are mapped to classes and object properties, respectively. In addition, OWL has many profiles and sublanguages, some

[6] https://www.w3.org/TR/rdf11-primer/#section-triple
[7] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/
[8] https://www.w3.org/2007/OWL/draft/ED-owl2-manchester-syntax-20081128/

of which map to specific DL languages targeting a specific range of applications. See Table 2.10 for a comparison of OWL and DL terminology.

Table 2.10: A mapping of some OWL and DL terminology.

Web ontology language **(OWL 2 DL profile)**	**Description logic** **(DL)**
Ontology	Knowledge base
Individual	Instance
Class	Concept
Owl:thing (top class)	Top concept ($\top$)
Owl:nothing (bottom class)	Bottom concept ($\bot$)
Class expression	Concept definition
Object property	Role
Data property	Concrete role

In OWL, restrictions can be applied to roles (known as "object properties" in OWL), where the subject, and/or the object of a role can be constrained to specific individuals belonging to either a simple or complex class.

Furthermore, OWL 2 DL has three profiles (or sublanguages). The first is the OWL EL profile, which is based on $\mathcal{EL}^{++}$ (extensions of the $\mathcal{EL}$ language). It is suitable for applications that have less expressivity requirements and that have a large number of concepts and roles, and it is frequently used in the medical field for various reasoning tasks. The second is OWL QL, which is based on fewer expressive DL languages and which mainly focuses on query answering applications against a large number of assertions. This OWL profile can be built on top of relational databases. The third is OWL RL, which focuses on addressing scalability in DL reasoning through the use of rule-based reasoning engines with slightly reduced expressivity. All three OWL profiles (EL, QL, and RL) are subsets of the OWL 2 DL, which has the more expressive DL languages (e.g. $\mathcal{SROIQ}^{(\mathcal{D})}$). The DL language of this project, $\mathcal{ALCQ}^{(\mathcal{D})}$, is mapped into the OWL 2 DL profile, which can use its expressive reasoners such as HermiT. The description of the OWL 2 architecture, its sub profiles, and a mapping of its sublanguages to DL can be seen in Figure 2.14 and Figure 2.15, respectively.

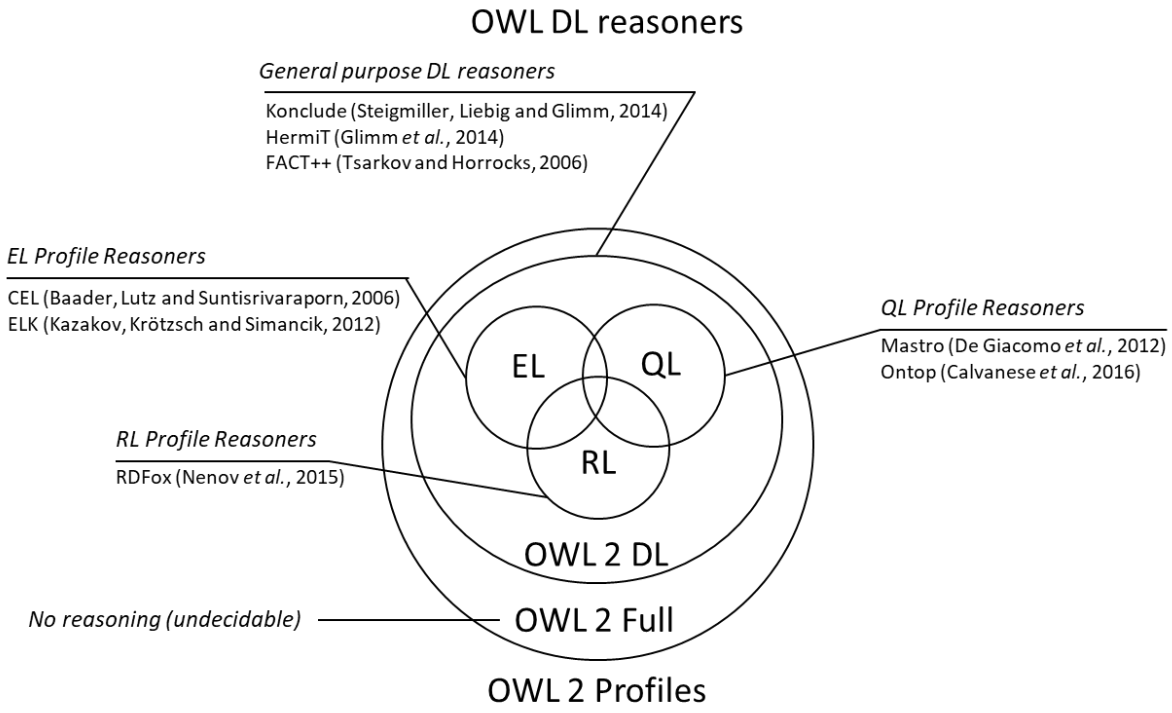


Figure 2.14: An overview of OWL 2 profiles and some examples of their reasoners.

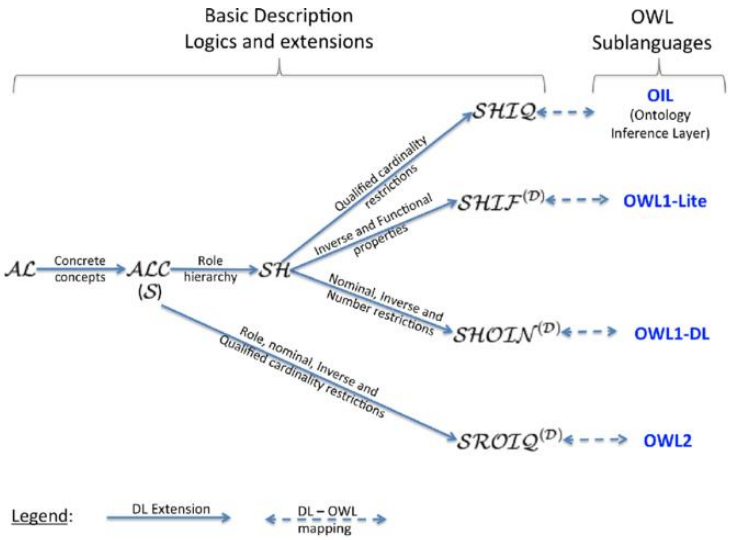


Figure 2.15: A mapping of OWL profiles and their corresponding DL languages (Petnga and Austin, 2016).

Both OWL and DL use the open-world assumption (OWA), which assumes knowledge to be semantically incomplete: If something cannot be proved true or false by the current ontology, then the reasoning result would be "possible" (or simply "I do not know"). In the closed-world assumption (CWA), on the other hand, if the result cannot be proved to be true, then it is false (e.g. relational database systems). In addition, both OWL and DL do not follow the unique name assumption (UNA) –

that is, a single entity in OWL and DL can have multiple names (or identities). See Table 2.11 for a comparison of closed-world and open-world data representations.

Table 2.11: A comparison of open-world and closed-world representations.

	Closed world	**Open world**
Data representation example	Relational database system (e.g. mySQL)	OWL ontology (e.g. Protégé[9])
Data representation	Database tables	Model (interpretation)
Querying/reasoning	SQL	SPARQL (instance checking)
Query answering outcome	Yes, no	Yes, no, "maybe"
Assumptions	UWA, UNA	OWA

It is worth making such a distinction because a certain world representation may not be suitable for all applications. For example, closed-world representation is mostly used for data storage and retrieval, such as representing customer purchases in an online store. Open-world representation is typically used when reasoning tasks are needed, such as proving or disproving something based on a given knowledge base.

In the context of ILP learning, the world assumptions (closed or open) affect how a hypothesis is evaluated. In closed world assumption, given the set of all individuals I in a knowledge base (ABox in particular), the set of positive examples P (a set of individuals, a subset of I), and the set of negative examples N (a set of individuals, also a subset of I); evaluating a hypothesis H against a knowledge base (the set I in particular), will yield R the set of individuals covered by H against I. The covered positive and negative examples, are the intersection sets $R \cap P$ and $R \cap N$ respectively. The number of individuals in $R \cap P$ is the number of covered positives, and in $R \cap N$ is the number of covered negatives.

In open world assumption, a hypothesis is evaluated similarly to closed world assumption, however, with key differences. In closed world, instance checking is typically sufficient for hypothesis evaluation, whereas in open world, a DL reasoner is typically used that will use instance checking and possibly in combination with other reasoning tasks (e.g. realization) to infer new individuals (through open world reasoning) that may be considered as covered examples.

[9] https://protege.stanford.edu/

Hypothesis evaluation in open world assumption is more expensive than closed world assumption due to using more reasoning tasks (realization and instance checking) instead of only instance checking (for closed world).

In terms of closed and open world ILP learning in DL, closed world ILP learning will result in a single hypothesis that (ideally) cover all positive examples and none of the negative examples. In contrast, open world learning will result in two DL hypotheses, a positive hypothesis that should cover all positives and no negatives, and a negative hypothesis that cover all negatives and no positives; that is, a new individual is not considered negative if it is not covered by the positive hypothesis, for the individual to be considered negative, it has to be covered by the negative hypothesis and vice versa. In the context of this thesis, the focus is on closed world DL-based ILP learning.

There are several scalable data projects in the field of OWL and DL; these projects extend and semantically link the web data. In that effort, a combination of automated and manual approaches were used to collect and combine datasets and ontologies from various sources into a semantically central massive knowledge base that captures a massive amount of human knowledge in a machine-readable format. These massive knowledge bases can be used for a variety of query answering and reasoning applications. A notable example in the effort of massive knowledge base construction is the linked open data (LOD). In LOD, hundreds of knowledge bases and datasets, such as DBpedia, YAGO,[10] GeoNames,[11] and WordNet[12] are combined. DBpedia is an RDF dataset constructed from Wikipedia articles, and it encompasses a wide variety of human knowledge. Similar to DBPedia, YAGO (or Yet Another Great Ontology) also collect data from Wikipedia; however, YAGO is more recent and aims for more data accuracy. GeoNames is a geographical dataset providing geographical information about places. WordNet is a dataset that lexically links words in English. These datasets form a massive graph of relations to provide a machine-readable format of human knowledge. See Figure 2.16 for the massive LOD's graph.

[10] https://yago-knowledge.org/
[11] https://www.geonames.org/
[12] https://wordnet.princeton.edu/

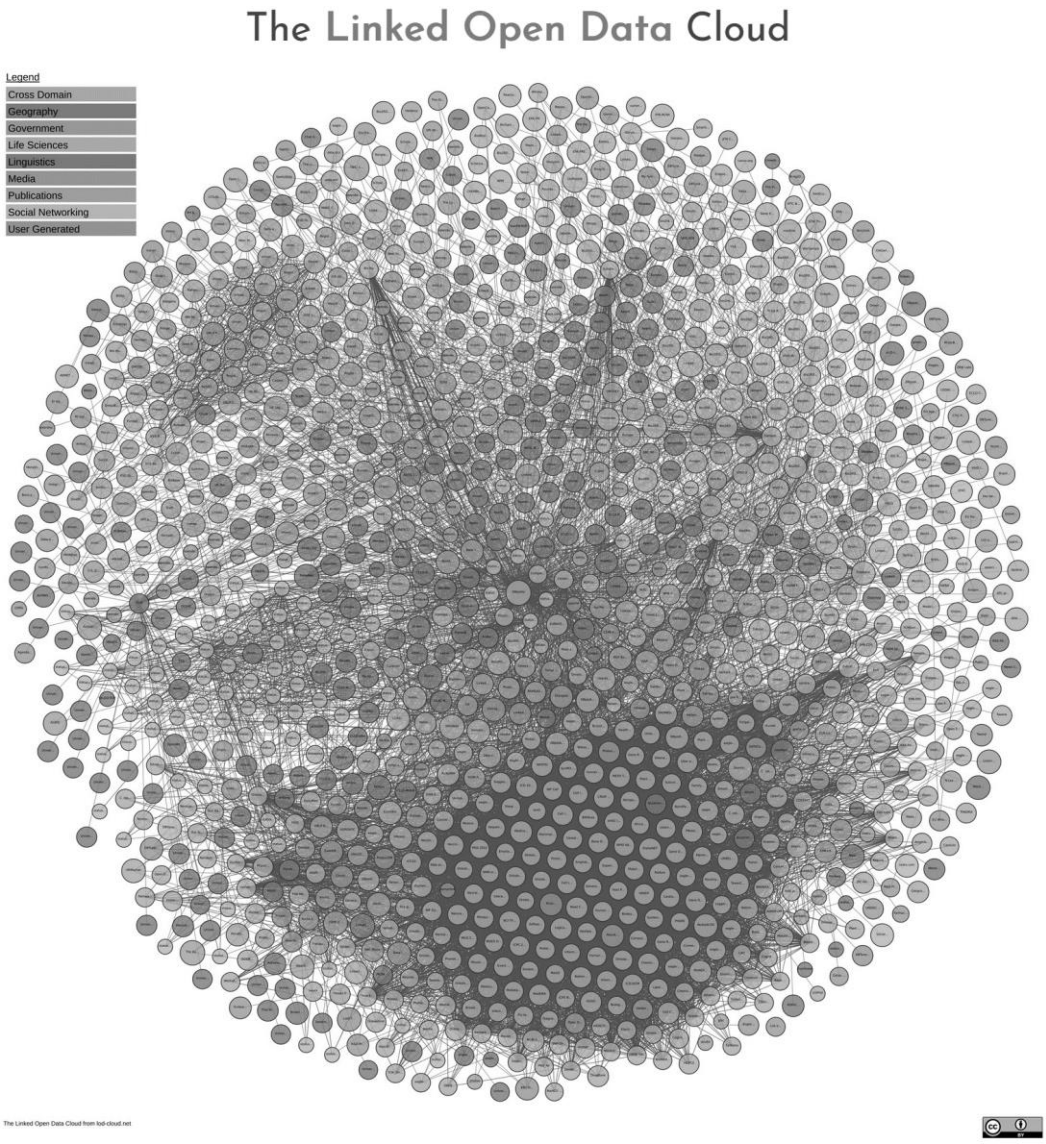


Figure 2.16: An overview of the (LOD) cloud graph.[13]

The discussion of the LOD is made to support two key issues. First, it communicates the availability of massive real-world DL datasets. Second, it supports the idea that solving the scalability issues of ILP learning in DL opens a new horizon, through which ILP learning from massive DL data is possible.

2.3 Parallel Computing

Parallel computing (PC) carries the potential to address ILP scalability issues, especially in the specialized context of this work. Therefore, the background of parallel computing and its related concepts is provided. Parallel computing refers to the ability to do multiple computing tasks in a parallel (simultaneous) manner. There are many classifications of parallel computing – for example, based on the computing architecture, application type, and memory layout (and usage). In terms of computing

[13] https://lod-cloud.net/

architecture, there are four types: SISD, SIMD, MISD, and MIMD. The first, SISD, is the only non-parallel architecture. See Table 2.12 for Flynn's taxonomy of computer architectures.

Table 2.12: Flynn's taxonomy of computing architectures (Flynn, 1972).

Hardware Architecture	**Example (s)**
Single instruction single data (SISD)	• Single core CPU
Single instruction multiple data (SIMD)	• Graphical processing units (GPUs) • Vector processors • CPUs with SIMD instructions
Multiple instruction single data (MISD)	• Safety-critical systems
Multiple instructions multiple data (MIMD)	• Multi-core CPUs

Parallel computing is mostly employed for speeding up and increasing the efficiency of computations. However, in some applications (e.g. safety-critical systems), MISD is used to ensure that a system is producing the correct result. For example, two separate processors may use two different algorithms to produce the output for the same input data. The outputs from the algorithms are then compared; if they agree, the system is performing correctly; otherwise, a fault is detected. In recent years, many multi-core CPUs have included instructions that support SIMD (vector) operations and that combine both MIMD and SIMD in the same hardware. In the review that follows, we provide a detailed discussion of GPUs and multi-core CPUs. Before that, let us first look at the types of parallel memory architectures as well as the two forms of parallel computing: task and data parallelism.

2.3.1 Parallelism memory architecture

In parallel computing, there are two types of memory layouts (or architectures). These architectures affect the performance of parallel computing in many aspects, such as speedups (compared to sequential execution), processing and memory scalability, and cost.

Shared memory architecture

Shared memory architecture refers to when multiple processors (in a single chip or across multiple chips) share a single memory. There are two variations of shared memory parallelism. The first, known as uniform memory access (UMA), is when multiple processors (or cores) share a single memory by a single bus connection. The second type is when several UMA multiprocessors are connected to each other through multiple buses connections, which is known as non-uniform memory access (NUMA). See Table 2.13 for a comparison of the two shared memory architectures. See also Figure 2.17 and

Figure 2.18 for a visualization of both variations. In a shared memory parallelism, the bus-connected processors can be a heterogenous mix of CPUs and GPUs.

Table 2.13: A comparison of UMA and NUMA shared memory architectures.

Shared memory variation	Examples	Advantages	Limitations
Uniform memory access (UMA)	• Single multi-core Processor • Multiple (multi-core) processors • CPU(s) and integrated GPU (using the same memory)	• Simplicity • Equal memory access speed	• Memory scalability issues • Limited memory bandwidth issues (central shared bus)
Non-uniform memory access (NUMA)	• CPU(s) and GPU(s) with its own dedicated memory • High performance servers	• High total memory bandwidth (due to several buses)	• Uneven memory access speed • Costly when compared to UMA

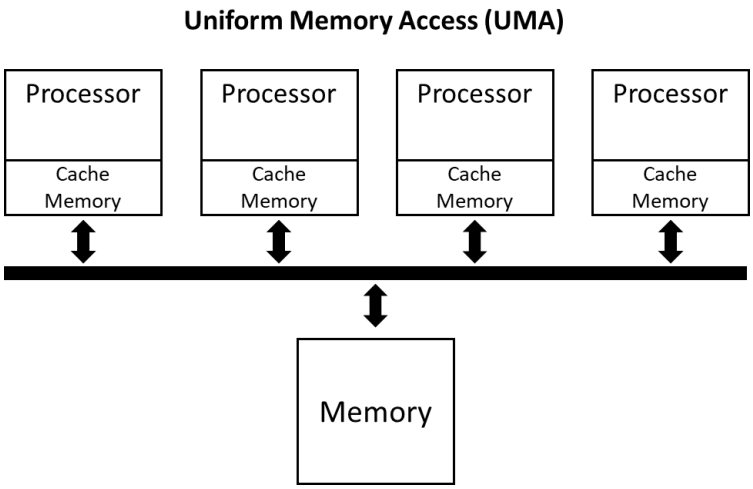


Figure 2.17: An overview of the UMA type.

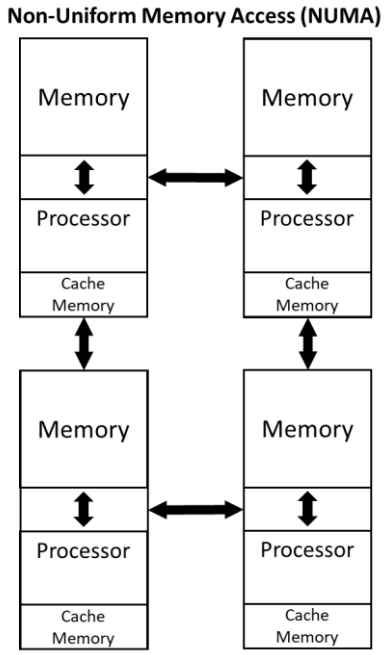


Figure 2.18: An overview of the NUMA type.

Distributed memory architecture

In the distributed memory architecture, the processors are connected like NUMA; however, the connection is made through a network (e.g. TCP/IP). Moreover, unlike NUMA, the distributed processors cannot directly access each other's memory. In terms of the network connection, the distributed processors can be connected either in a single local network or within a larger network (e.g. the internet). See Figure 2.19 for an overview of the distributed memory architecture.

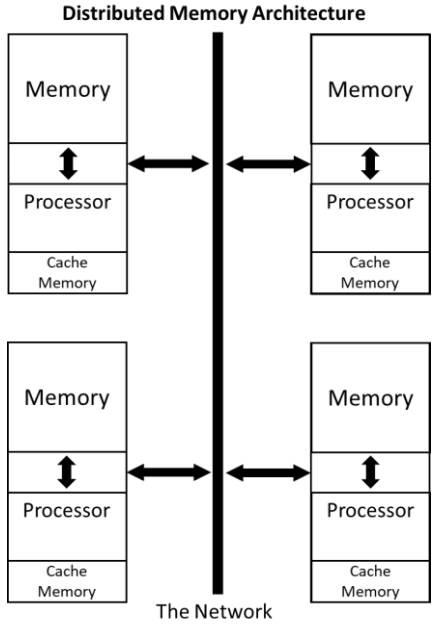


Figure 2.19: An overview of the distributed memory architecture.

Hybrid memory architecture

Both memory architectures discussed above have advantages and limitations; see Table 2.14 for a comparison of the shared memory and distributed memory architectures.

Table 2.14: A comparison of the shared and distributed memory architecture.

Memory architecture	Advantages	Limitations
Shared memory	• Software development simplicity • Fast memory access	• Can be costly to scale processing power – may replace existing hardware to accommodate new processor(s) • Limited memory (space) scalability – due to hardware (connection) limitation
Distributed memory	• Scalable processing power and memory space • Simple to scale – simply add new processor(s) to the network	• Work distribution and load balancing complexity • Network communication overhead • Uneven memory access speed (local vs. remote network data access) • Software development complexity

A hybrid architecture could maximize the advantages from both architectures and minimize their limitations. For example, an application may initially work only on shared memory parallelism. When more processing power is needed, the application may use other (distributed) processors that exist in the network. See Figure 2.20 for an overview of the hybrid architecture. In Figure 2.20, the distributed processors are connected through the network, and the multiple processors (GPUs and/or CPUs) that exist in a single machine are bus-connected via UMA or NUMA.

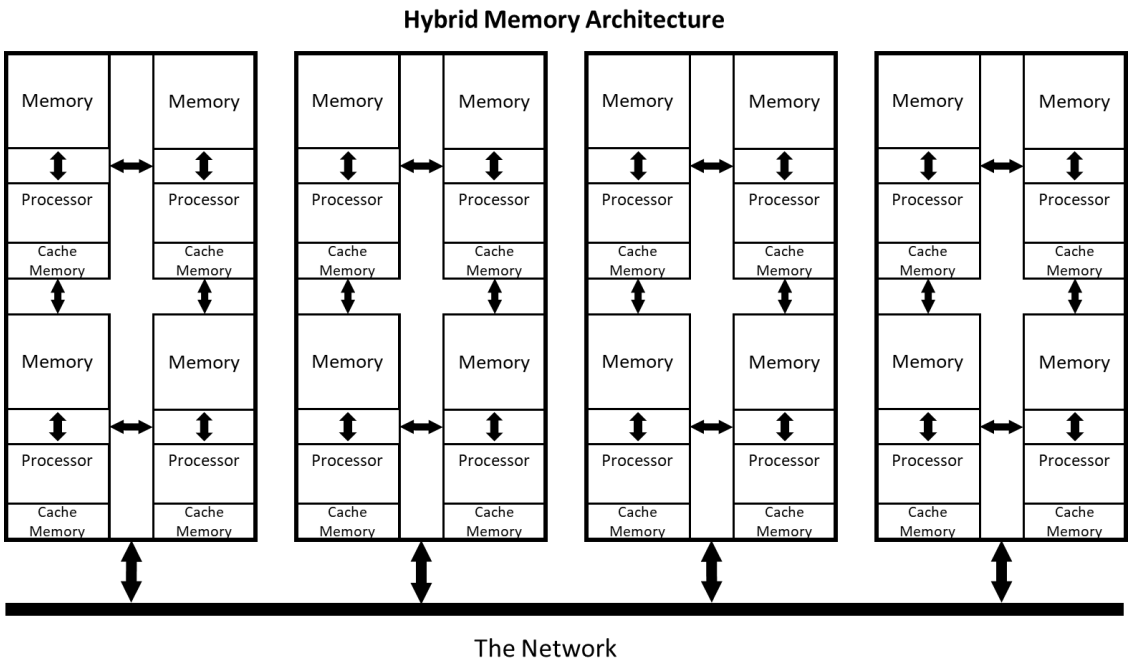


Figure 2.20: An overview of the hybrid memory architecture.

2.3.2 Task parallelism

Task parallelism refers to the ability to carry out multiple tasks independently and simultaneously by distributing these tasks across multiple processors. Task parallelism can be observed in many computing applications. Operating systems run user programs and manage computer resources in parallel through multiple processors. Web servers apply task parallelism, with which multiple users (or clients) can be served by the same server simultaneously. Task parallelism is also observed in web browsers, where the user browses the web and downloads files at the same time. Task parallelism can take place at the operating system level through multi-processing and at a program level through multithreading.

2.3.3 Data parallelism

Data parallelism refers to the ability to simultaneously apply the same computational operation to multiple data items. A typical example of data parallelism is the sum of two arrays (A and B) into another array (C). Data parallel processors are known as vector processors, which follow the SIMD architecture by simultaneously applying the same operation to multiple data items using a single processor instruction. Vector processors are the opposite of scalar processors (i.e. SISD or single core CPUs), with the latter only being able to process one data item at a time. See Figure 2.21 for a comparison of scalar and vector processors for adding two arrays using a for-loop.

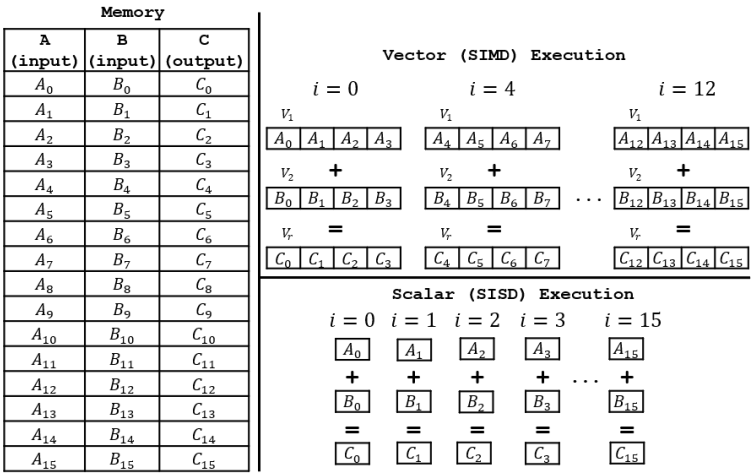


Figure 2.21: A comparison of scalar and vector processors at adding two arrays (*A* and *B*) into *C*.

In Figure 2.21, *i* refers to the array index. In the scalar processor, only one array index (or a single row) is processed at a time. However, in the vector processor, multiple rows (or indexes) are processed (computed) at the same time using single CPU instructions; see Figure 2.22 for the pseudocodes of both execution models.

```
[Scalar Execution]
For i=0 to i=15
  1 Read A_i
  2 Read B_i
  3 Add A_i and B_i into a register
  4 copy register value into C_i
  5 Increment i by 1
```

```
[Vector (SIMD) execution]
For i=0 to i=15
  1 Read 4 values starting from A_i into vector V_1
  2 Read 4 values starting from B_i into vector V_2
  3 Add V_1 and V_2 into vector V_r
  4 Copy V_r into [C_i:C_i+3]
  5 Increment i by 4
```

Figure 2.22: A comparison of scalar and vector processing pseudocodes.

In Figure 2.22, each step (operation) within the for-loop body are assumed to take a single processor instruction. In SIMD, a single instruction is required to load multiple values, and another single instruction is needed to compute the result for these values; therefore, a speedup is achieved as a consequence. For the computation examples in Figure 2.21 and Figure 2.22, the vector model achieves speedup four times greater than the scalar one.

Depending on the data parallel hardware used, the processing unit can be either a single vector processor, or a set of vector processors. The number of values that can be used in a vector is known as the vector length. A well-known data parallel hardware is the graphical processing unit (or GPU). The GPU can have multiple vector processors that are capable of computing results for huge arrays and matrices. GPUs were initially intended for accelerating graphics-related computations. In recent years, GPUs have been successfully used for accelerating other computations – hence the term GPGPU (University of Edinburgh, 2020) (short for general-purpose GPU) refers to the use of GPUs to accelerate non-graphics computations.

Given the same cost, data parallelism hardware (e.g. GPU) outperforms task parallelism hardware (e.g. multi-core CPU) because of its simpler manufacturing complexity. In terms of manufacturing costs, adding a set of parallel ALUs (for data parallelism) is much simpler and cheaper than adding CPU cores. Typical data parallel hardware can only be used to accelerate computations by acting as coprocessor. Task parallel hardware is able to simultaneously process multiple independent tasks, with its cost being its limitation. In many real-world applications, task and data parallelism are usually combined. In fact, many modern processing units (e.g. modern multi-core CPUs) have both the hardware (in a single chip) for data parallelism (through SIMD instructions) and the multiple cores (for task parallelism). An example software that exploits modern (multi-core) processers and GPUs is simulation software. In such software, the GPU is used for computing the simulation results in real time, and the CPU cores handle the input/outputs of the simulation, which can also use its SIMD instructions for light data parallelism. It is worth noting that task and data parallelism are not bound to a particular hardware or a particular library (API). Task/data parallelism is more of a view of how parallel computing is employed to solve parallel computing problems. In this project, we adapt both task and data parallelism to accelerate both ILP hypothesis search and evaluation via the use of GPUs and multi-core CPUs. In the following sections, GPGPUs and multi-core CPUs are discussed in more detail given their direct relevance to this work.

2.3.4 General-purpose GPU

A GPU was originally designed for accelerating data parallel computations in computer graphics. However, in recent years, GPUs have been used (through GPGPU) in many applications (Nvidia, no date b), and they have managed to achieve impressive speedups. Even though GPUs offer impressive computing power, they require significant attention to algorithmically formulate a computation problem in a way that can be executed in the GPU hardware. In GPUs, the data for both input and output are typically represented as a 1D, 2D, or even 3D matrix. As shown in Figure 2.23, there could be a single matrix or a set of matrices.

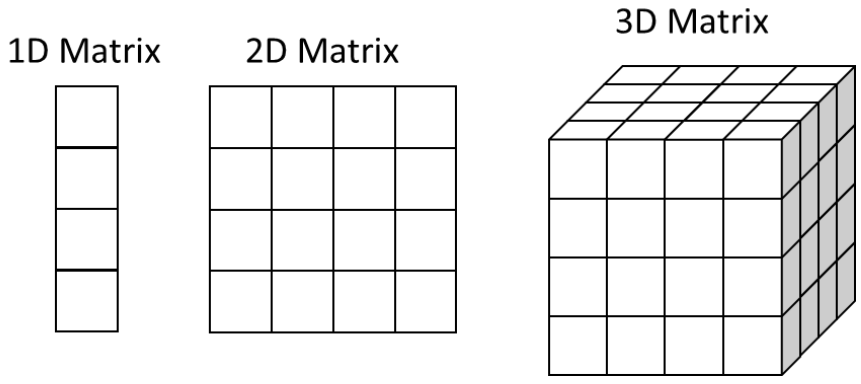


Figure 2.23: GPU's data representation matrices.

Workflow of GPU-accelerated program

In a typical GPU-accelerated program, the program runs in the CPU. When the GPU computation is needed, the CPU dispatches the computation task to the GPU. The GPU then executes its computationally intensive data parallel task. During the GPU execution, the CPU can either wait for the GPU to finish or do other task. Once the GPU finishes its assigned task, the CPU then uses the results returned by the GPU in the rest of the application. See Figure 2.24 for an overview of a GPU-accelerated program.

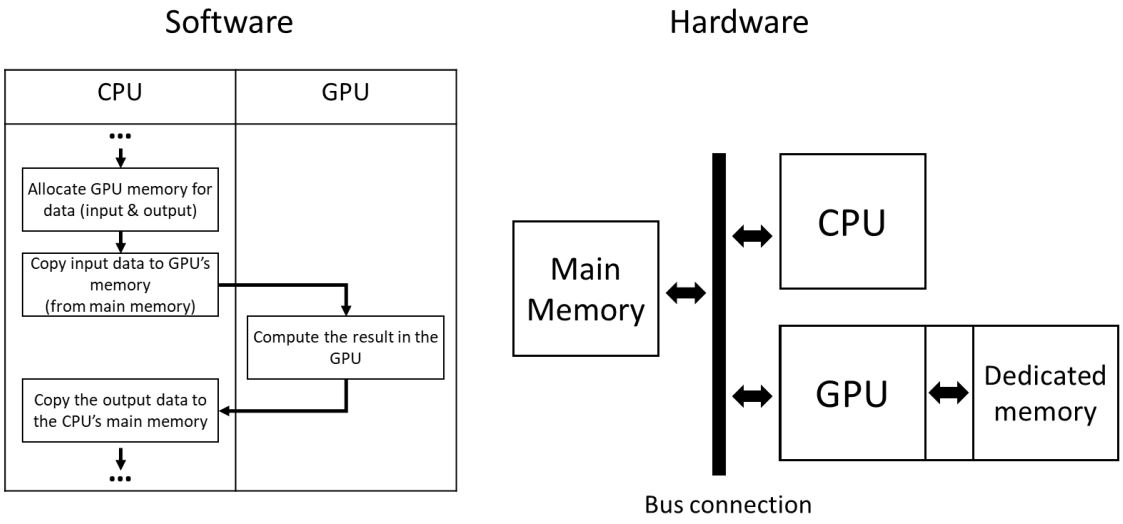


Figure 2.24: An overview of a typical GPU-accelerated program.

In Figure 2.24, the GPU-accelerated program allocates GPU memory to store the input and output data (i.e. matrices). Thereafter, the program copies the input data from the main memory into the GPU memory. Once the GPU has the input data in its memory, the CPU signals (dispatches) to the GPU to execute its parallel task. Once the result is computed by the GPU, the CPU copies the result (i.e. the output) from the GPU memory into the main memory for further processing. The details of the GPU architecture are explained next.

The GPU architecture – CUDA perspective

A GPU is essentially a set of vector processors (known as streaming multiprocessors, or SMs). Each SM may contain one or more streaming processors (CUDA cores, or simply ALUs); CUDA stands for compute unified device architecture. In a GPU context, a single core is a single ALU, unlike in CPUs, where a single core is a full processor. In terms of hardware implementation, as with CPUs, GPUs also have different hardware implementation architectures, which do not affect their conceptual operation (i.e. different implementation architectures aim to improve the hardware performance). For example, a GPU architecture may add more streaming processors, while others may have fewer streaming processors, which run at higher clock speeds. An example of a single streaming processor in a GPU with Nvidia's Fermi architecture can be seen in Figure 2.25.

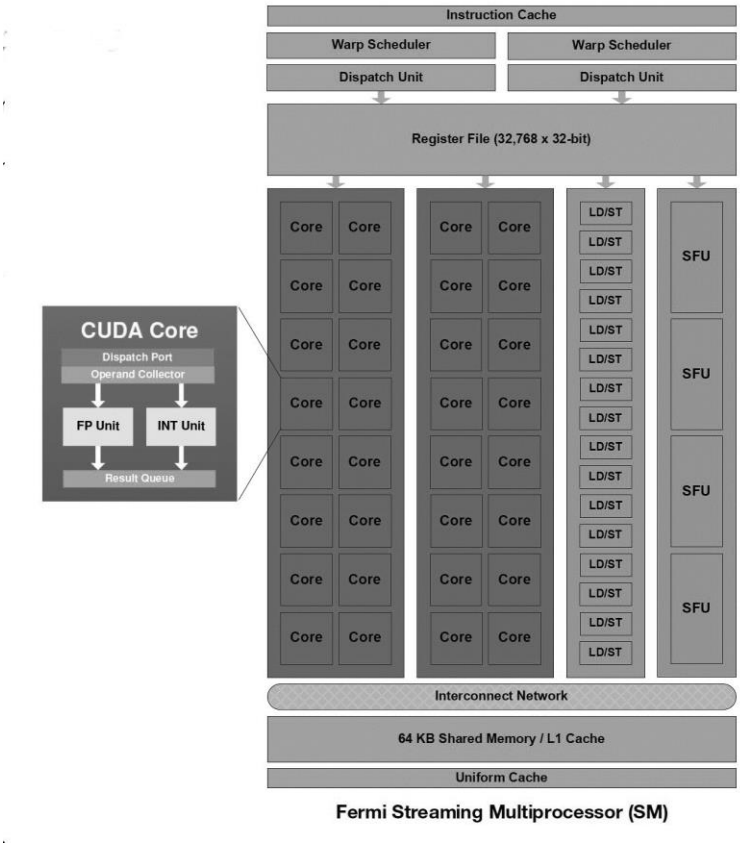


Figure 2.25: A single SM with its CUDA cores (Nvidia, 2009).

In Figure 2.25, a single Fermi SM contains a total 32 ALUs (CUDA cores), 32,768 four-byte registers, and two warp schedulers. Warp schedulers are hardware schedulers responsible for work management and distribution among CUDA cores within a single SM. A single CUDA core is capable of computing operations on 32-bit integers and floats, although 64-bit operations are also supported. The 16 LD/ST units are used for load/store memory operations. With regard to the four special function units (SFUs), a single SFU is responsible for computing functions like trigonometric functions, such as sine and cosine. Other GPUs may have variations of these hardware units depending on their hardware implementation.

Regardless of a GPU hardware implementation, in GPGPU, a dispatched task to the GPU is known as a kernel. A kernel is small piece of code (e.g. C language function) that applies mathematical operations in parallel to one or more matrices. The GPU handles all the work distribution and load balancing through a hardware scheduler. In executing a GPGPU kernel, a single grid is used. A grid contains a set of thread blocks, and each thread block is comprised of a set of threads (which is the smallest unit in the GPU). A single thread only processes a single matrix element (or a single array index). The threads in a thread block are grouped into warps (32 threads per a single warp) and then executed in parallel in a lock-step manner; all the threads in a single warp execute the same instruction (but on different array/matrix indexes). Depending on the thread block size (i.e. number of threads), a single warp or multiple warps may be needed to execute the block's threads (within the same SM); that is, all threads

that belong to the same block are executed on a single SM. When the CPU dispatches a computing task (kernel) to the GPU, it will also include its grid size (number of thread blocks) and the block size (number of threads in a block). The GPU will use these two parameters for its work distribution and load balancing (among its SMs). The GPU may have multiple hardware warp schedulers per single SM. These warp schedulers keep their corresponding SMs busy by scheduling a constant supply of threads to be executed (through a set of warps) to maximize efficiency. The process of warp scheduling will continue until all the threads in the kernel's grid are executed. See Figure 2.26 for the GPU thread hierarchy.

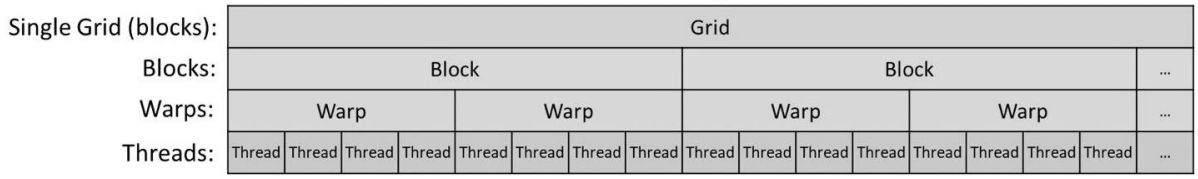


Figure 2.26: The GPU thread hierarchy.

The GPU memory architecture

A GPU has different kinds of memory that directly affect their performance. In other words, careful attention must be made when formulating a computation problem in order for the GPU to avoid poor GPU performance. As a result, different types of GPU memory are discussed.

There are two kinds of GPUs in terms of their memory. The first kind is integrated GPUs, which use the main memory (RAM) for data storage and usually have less computational power than the GPUs of the second kind. The second type of GPU has its own dedicated memory and can also still access the main memory; they tend to be used in most GPGPU computations because they are more computationally powerful. However, GPUs of the second type tend to be more expensive than the integrated ones.

From the CUDA perspective, there are different types of memory within a GPU. The first is the GPU's dedicated global memory, which is the largest (in ~GBs) but slowest memory. This memory typically stores the input and output matrices. The second memory type is the constant memory, which is good for constant read-only data lookups; it can be as fast as reading a register because of caching. However, it is small in size (a few KBs). The third, texture memory, is more of a read-only memory access optimization for 2D matrices stored in the global memory. The fourth, shared memory, provides shared data storage for threads within the same block, and it has a small size (a few KBs). Finally, the local memory and the registers are only visible to a single thread. The registers are the fastest memory type in the GPU, and a single register holds only 4 bytes of data. When all registers are used (i.e. there is no available register for the kernel), the thread's local data storage will leak into the local memory. The local memory is an abstraction of the global memory; however, it is only visible to its corresponding threads, and it is as slow as the global memory. A summary of the GPU memory types can be seen from

Table 2.15. An overview of the GPU memory types with their thread hierarchy can be seen in Figure 2.27. In addition, see Table 2.16 for a mapping between the CUDA API and the OpenCL API.

Table 2.15: The GPU memory types.

Memory Type	**Access Speed**	**Operation Type (Read/Write)**	**Memory Size**
Global memory	Slowest	Read/Write	Usually in GBs
Texture memory	Faster than global memory	Read only	10,000s of array indexes
Constant memory	Faster than texture memory	Read only	Few KBs
Local memory	Same as global memory	Read/Write	Several KBs
Shared memory	Comparable to registers speed	Read/Write	Few KBs
Register memory	Fastest	Read/Write	4 bytes (32 bits)

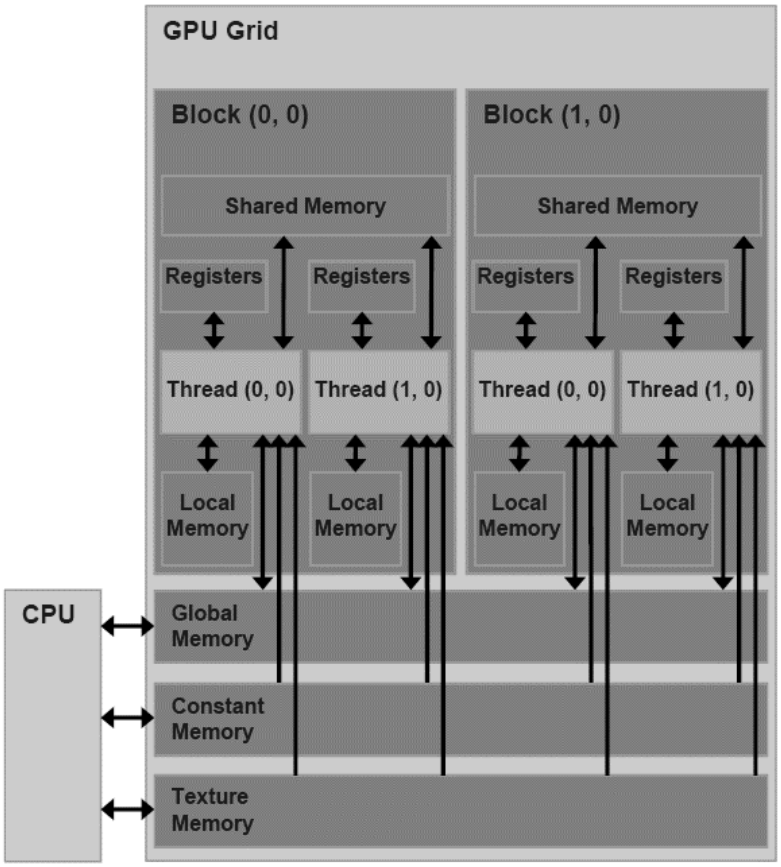


Figure 2.27: An overview of the GPU memory types and their thread hierarchy (Nvidia, 2007).

Table 2.16: A comparison of OpenCL and CUDA.

	Open Computing Language (OpenCL)	**Compute Unified Device Architecture (CUDA)**
Supported hardware	Brand agnostic (open Standard)	Brand specific
Supported processor types	(Multi-core) CPUs and GPUs	Nvidia GPUs only
Architecture and execution model terminology	Compute Unit	Streaming multiprocessor
	Processing Element	Streaming processor (CUDA Core)
	Kernel	Kernel
	NDRange	Grid
	Work-group	Block
	Wavefront (Or Subgroup)	Warp
	Work-Item	Thread
Memory model terminology	Global memory	Global memory
	Global memory	Texture memory
	Global memory	Local memory
	Constant memory	Constant memory
	Local memory	Shared memory
	Private memory	Registers

2.3.5 OpenMP (Open Multi-Processing)

OpenMP (Open Multi-Processing) is a multithreading library for shared-memory multi-processing and is commonly used with C/C++ and Fortran (OpenMP, no date). A review of OpenMP is necessary because this project employed parallel computing of shared-memory CPUs through the OpenMP API to accelerate parts of ILP computations (i.e. chapter 6). The alternative for using OpenMP, is to directly

create and manage threads, including the complexity of managing their execution and work distribution. Hence, OpenMP is used instead since it hides the unnecessary complexity of threads management. OpenMP follows the fork-join model of parallel computing, as shown in Figure 2.28.

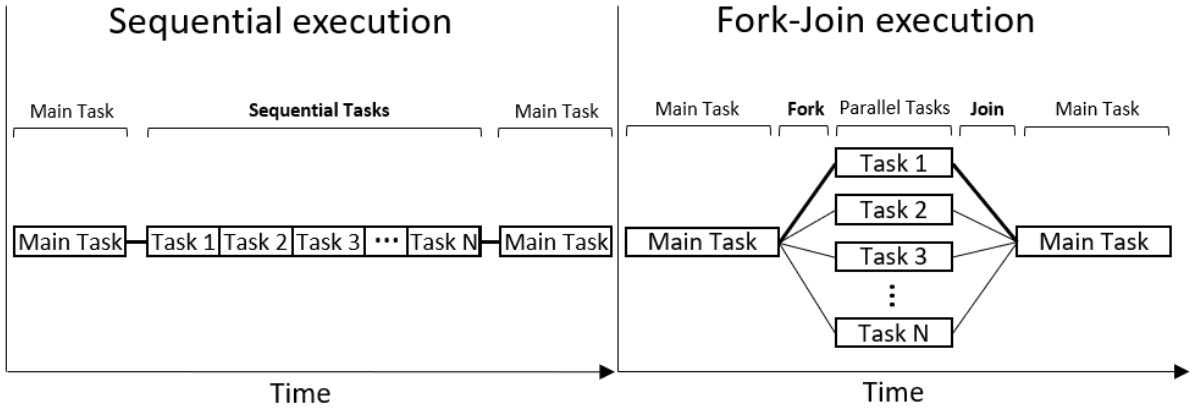


Figure 2.28: Sequential execution versus OpenMP's fork-join model.

In Figure 2.28, it is easy to observe the difference between the sequential execution model (SISD) and OpenMP (the fork-join model). In the sequential execution, all tasks are executed sequentially. While in OpenMP, the main task first executes in a single thread before it branches out into multiple parallel threads, each of these parallel threads executes a different task independent of the other threads. After all the threads finish their execution, the control flow returns to the main task (thread).

The OpenMP library handles the details of work distribution and load balancing for the user. It also handles the creation and destruction of threads. OpenMP maintains the threads in a thread pool to reduce the expensive cost of frequently creating and destroying threads. Once a thread is created, it remains in the OpenMP thread pool until the program terminates. The user can configure many aspects of the library, such as the number of used threads, the work distribution, and the load balancing policy between the threads. Moreover, the user can also configure the threads assignment to the physical processors or cores. OpenMP handles the implementation details of these settings. See Figure 2.29 for a comparison of a sequential C/C++ program and its OpenMP multithreaded counterpart on 16 threads.

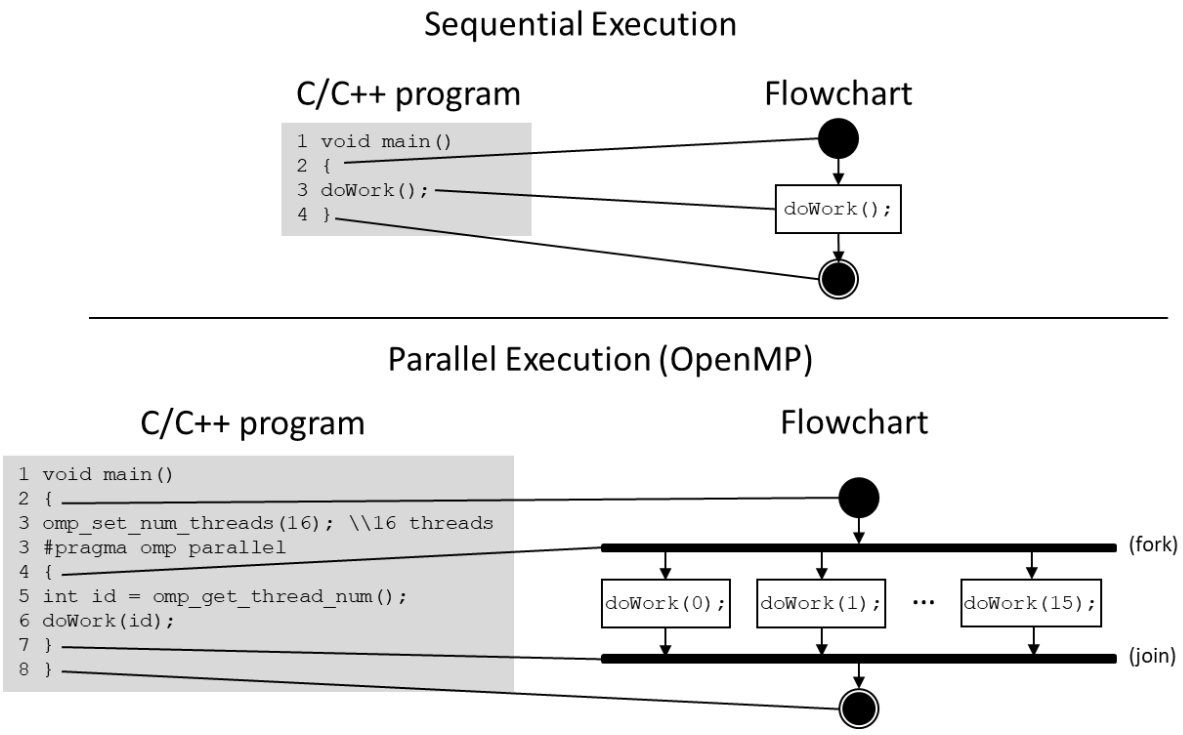


Figure 2.29: A sequential C/C++ program and its parallel OpenMP counterpart.

In Figure 2.29, the fork-join parallel approach can be clearly seen. In OpenMP, the same code is assigned to the forking threads, each with a unique thread number (or ID). OpenMP can be used for both task and data parallelism.

Task parallelism in OpenMP

For task parallelism in OpenMP, the thread number (ID) can be used to direct (map) a thread to a specific task, thus achieving task parallelism. See Figure 2.30 for an example of OpenMP task parallelism.

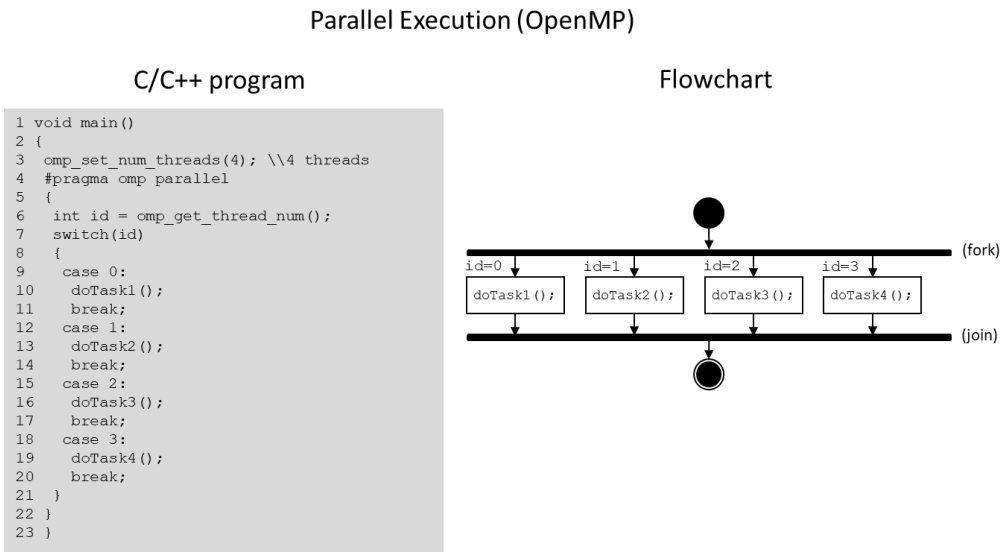


Figure 2.30: Task parallelism example in OpenMP.

Data parallelism in OpenMP

For data parallel applications, OpenMP can be used to distribute the iterations of a loop to the available processors/cores; see Figure 2.31 for an example of such data parallelism in OpenMP.

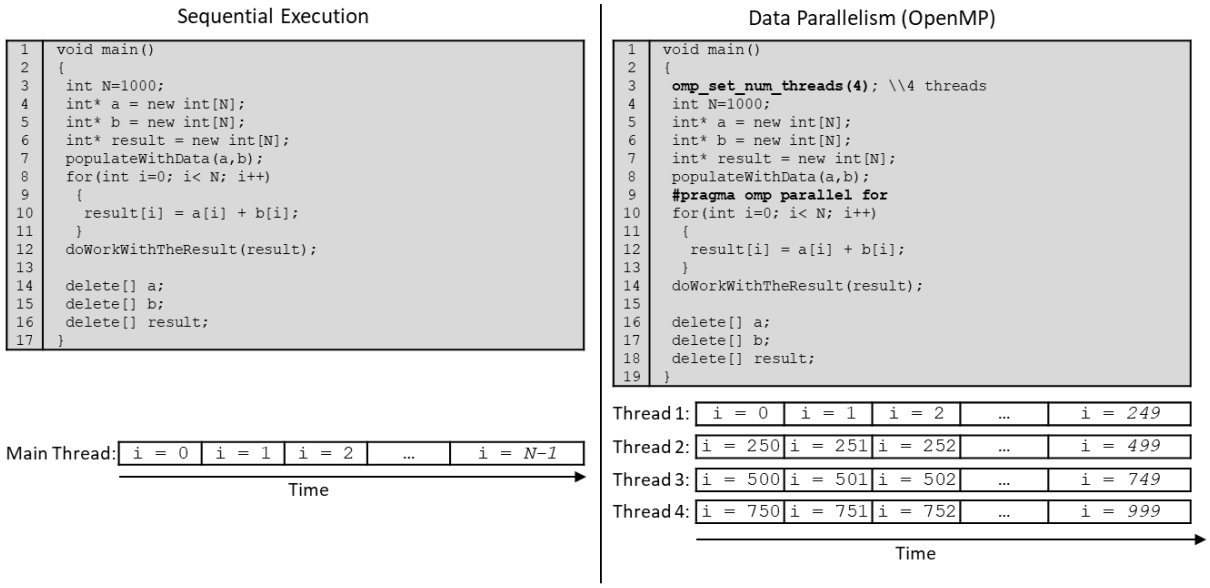


Figure 2.31: Data parallelism example in OpenMP (with default static scheduling).

In terms of parallelizing loops in OpenMP, there are several scheduling strategies. The first is the static (default) strategy, in which each thread processes a single predetermined chunk of loop iterations (See Figure 2.31). In the dynamic strategy, threads are assigned a chunk of loop iterations. Once a thread finishes its chunk, it is dynamically assigned another chunk. In the guided strategy, threads are scheduled like in the dynamic strategy; however, the chunk size (i.e. number of loop iterations) per thread may change. The guided strategy can be considered a special case of the dynamic strategy because it only adds the ability of dynamically varying the chunk size to enable better workload balancing. For a comparison of the loop scheduling strategies, see Table 2.17.

Table 2.17: OpenMP's loop scheduling strategies.

Loop Scheduling Strategy	**Advantage**	**Limitations**
Static	• Simple • Lowest scheduling overhead	• Potential workload imbalance (iterations may have variable computational complexity)
Dynamic	• Dynamic workload balancing	

	• Good for iterations that have different computational complexity	• Scheduling overhead
Guided	• Better dynamic workload balancing	

A key advantage of the OpenMP library is its handling of the technical (and implementation) aspects of work distribution and load balancing for the user; nevertheless, it still provides a sufficient degree of control and flexibility for shared memory multi-processing.

For a distributed memory environment, the MPI library is used. MPI (or Message Passing Interface) is a well-known communication protocol that governs the communication and the coordination among parallel processors (OpenMPI, 2020). MPI can be used on either shared or distributed memory environments, unlike OpenMP, which can only be used on shared memory environments. It is possible to combine MPI and OpenMP together in the same parallel system, by which OpenMP handles the parallel processors of a single machine and by which MPI handles the communication and management of the distributed machines. This combination achieves the hybrid memory environment.

There are libraries and APIs that can be used for GPGPU; two common APIs are CUDA (Nvidia, no date a) and OpenCL (Khronos Group, no date). The CUDA API is for writing and executing general purpose GPU code on Nvidia-branded GPUs. In contrast, OpenCL is a brand-agnostic API and an open standard by the Khronos Group for writing general data parallel code that can be executed on either GPUs or (multi-core) CPUs. There are also APIs that are built on computer graphics APIs, such as the ComputeShader in OpenGL (OpenGL Wiki contributors, 2019) and DirectCompute for DirectX (Microsoft, 2018). Both APIs (ComputeShader and DirectCompute) exploit the shader pipelines of their graphics APIs (DirectX and OpenGL) to run general-purpose GPU computations. Some data parallel APIs are used for massively data parallel applications, such as MapReduce (Dean and Ghemawat, 2008). In MapReduce, two functions are used – namely, *map* and *reduce* – for writing and executing massively data parallel computations that can span multiple (local and networked) processors.

Parallel computing provides a number of opportunities for speeding up computations in many domains, using a variety of hardware architectures and software APIs. A summary and a comparison of the discussed parallel computing APIs and their typical usage domain can be seen in Figure 2.32. For a summary and a mapping of parallel computing hardware and software APIs in terms of their parallel memory architectures, see Figure 2.33.

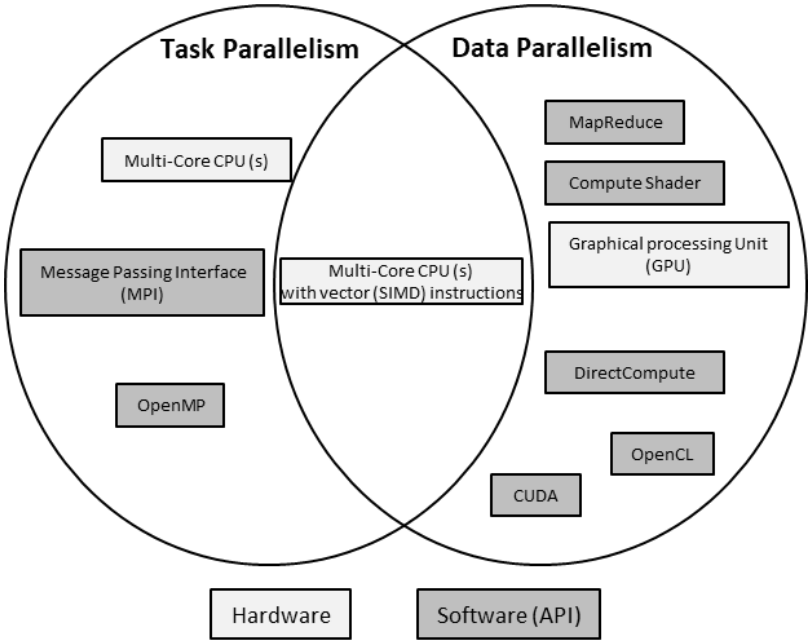


Figure 2.32: A summary and a mapping of the parallel computing technologies.

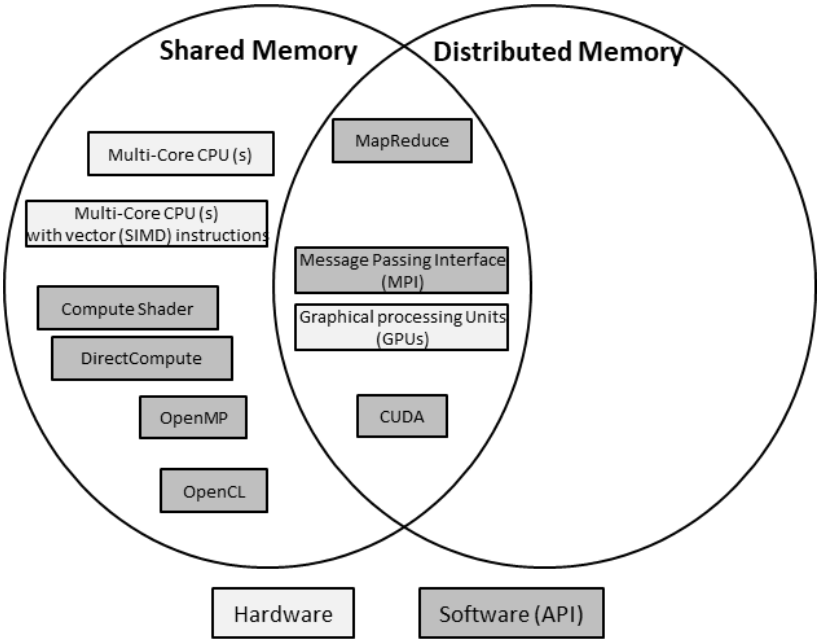


Figure 2.33: A summary and a mapping of parallel computing technologies based on their memory architecture.

2.4 Summary

The main purpose of this chapter is to provide the necessary background to understand this work.

A background in inductive logic programming provides an understanding of ILP learning as a machine learning technique and of its related factors. Inductive logic programming can use different logic formalisms for representing knowledge and models, such as PL, DL, and FOL, and each formalism has

its own degree of expressing power. However, using description logic in ILP learning provides the right balance between expressing power and computational complexity.

A background in description logic provides an understanding of the nature and characteristics of its knowledge representation, which was used as the knowledge representation for ILP learning in the context of this work. Description logic is a capable knowledge representation approach, and it has powerful syntax and semantics to capture and represent many complex, real-world knowledges, such as the DBpedia and the Linked Open Data projects.

A background in parallel computing and its related concepts provides the necessary understanding of the approaches and techniques useful for accelerating computations. Parallel computing provides the means and the approaches to accelerate computations by which one or more processors can simultaneously work together to solve a problem regardless of the underlying architecture of each processor (e.g. CPUs vs GPUs).

All the relevant concepts discussed throughout this chapter are summarised in Figure 2.34. In the next chapter, we discuss DL-based ILP learning.

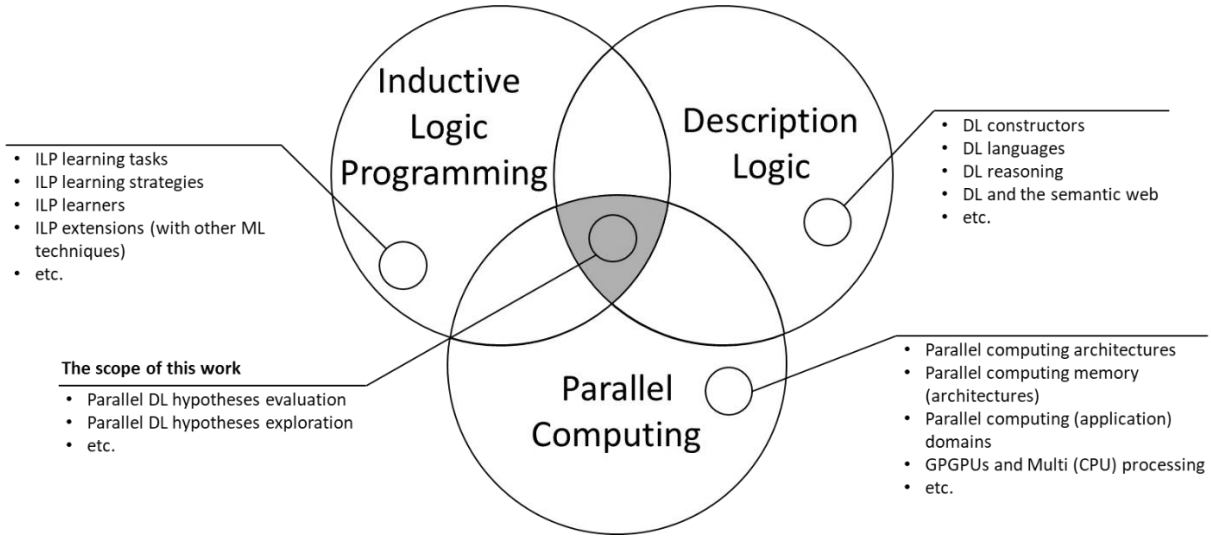


Figure 2.34: A summary of the relevant fundamental concepts.

Chapter 3: DL Inductive Learning (The DL-Learner)

In this chapter, we discuss the nature of DL-based ILP learning and some of its learning algorithms. Further discussion is made for the state of the art in DL inductive learning (i.e. the DL-Learner) because it forms the foundation of this work, including its search algorithm, refinement operator, and scoring function. The scope of this chapter lies in the intersection of inductive logic programming and description logic.

3.1 DL-based ILP learning

The nature of DL-based ILP learners is slightly different from classical ILP learners (which use horn clauses) because of the differences in knowledge representation. In DL, an OWL ontology is typically used to represent the knowledge base, including the learning examples. The learning examples are two sets of OWL individuals: one for positive examples and another for the negative examples. The output of ILP learning in DL is either one or more OWL class descriptions (or concept definitions in DL). DL-based ILP learners utilize DL reasoners to materialize the TBox of knowledge base. TBox materialization is the process of preparing the DL knowledge base for inductive learning by applying several DL reasoning tasks, such as realization to infer new assertions and classification to infer class hierarchy. In order to demonstrate ILP learning in DL, an example for learning the definition of a parent is given below. Consider the following DL knowledge base and the learning examples in Table 3.1, inductively learn (using ILP learning) the concept definition of a parent (i.e. $Parent \equiv ?$).

Table 3.1: An example DL knowledge base and its learning examples.

DL knowledge base	**Learning examples**
TBox: $Male \sqsubseteq Person$ $Female \sqsubseteq Person$ **ABox:** $David: Male$ $Mathew: Male$ $Mia: Female$ $Emma: Female$ $Liam: Male$	**Positive examples (parents):** {$David$, Emma} **Negative examples (non-parents):** {$Mathew$, Mia, Liam, Peter, Sophia, Lucy}

$Peter: Male$ $Sophia: Female$ $Lucy: Female$ $(David, Mathew): hasChild$ $(David, Mia): hasChild$ $(Emma, Mathew): hasChild$ $(Emma, Mia): hasChild$	

Before the DL learning starts, a DL reasoner is used to materialize the TBox in Table 3.1. The DL reasoner infers based on the TBox that, for example, $David$s an instance of $Person$ based on the TBox axiom $Male \sqsubseteq Person$. After the DL reasoning is completed, the ILP learning starts. The learning uses the set of parents (positive examples) and the set of non-parents (negative examples) to learn this definition: $Parent \equiv \exists hasChild(Person)$, which means that a parent is someone who has a child (from the $Person$ class). In the next section, we discuss some ILP learners in DL.

3.1.1 YinYang

(Iannone, Palmisano and Fanizzi, 2007) proposed a DL-based learner called YinYang (Yet another INduction Yields to ANother Generalization). YinYang employ both generalization and specialization approaches for learning DL hypotheses in the $\mathcal{ALC}$ language. In generalization (upward refinement), a disjunctive hypothesis is constructed iteratively using smaller sub-hypotheses (partial generalizations) that each cover some positive examples. After a sub-hypothesis is generated, and before it is added disjunctively (into the disjunctive hypothesis), it is checked first if it covers some negative examples. If the generated sub-hypothesis covers some negative examples, specialization (downward refinement) is used by constructing and using a *counterfactual* for that sub-hypothesis. A *counterfactual* (K) is a small hypothesis constructed from a sub-hypothesis ($ParGen$) and the negation of K is added conjunctively ($ParGen \leftarrow ParGen \sqcap \neg K$) in order to reduce the covered negatives for the given sub-hypothesis. A sub-hypothesis is specialized by conjunctively adding either a *counterfactual* or other small hypothesis (generated using different procedure from *counterfactual*). The Generalization step focuses on covering more positive examples, while the specialization step focuses on reducing the coverage for negative examples. The final learned DL hypothesis is a disjunctive hypothesis, a disjunction of conjunctions in some cases.

3.1.2 DL-FOIL

(Fanizzi, d'Amato and Esposito, 2008) proposed an adaptation of the classic ILP algorithm FOIL to learn DL hypotheses from DL knowledge bases (OWL-DL in particular). DL-FOIL follows OWA (the Open World Assumption) when ILP learning, with which it has the ability to directly learn from the inherently OWA DL knowledge bases, by using a DL reasoner for open world hypothesis evaluation. DL-FOIL learns a disjunctive hypothesis where each sub-hypothesis (in that disjunction) is a partial generalization (a hypothesis that cover some of the learning examples). In DL-FOIL, an upward refinement operator and a downward refinement operator are used. First, a partial generalization is generated using an upward refinement operator. If that partial generalization covers some negative examples, a downward refinement operator is used to specialize it to cover fewer negative examples, ideally to zero. If the partial generalization covers no negative examples either by default (when first generated) or by specialization (through downward refinement), that partial generalization is than added disjunctively as a sub-hypothesis in the final disjunctive hypothesis. The final (disjunctive) hypothesis is set to $\bot$ when the ILP learning is first started. The cycle of generating partial generalizations is then repeated and new sub-hypotheses are added (disjunctively) to the final hypothesis, where each time more positive examples are covered. A sub-hypothesis that cover some negative examples, will introduce its covered negatives into the final hypothesis as additionally covered negative examples. The ILP learning terminates when the disjunctive final hypothesis (ideally) covers all positives with few or zero negatives.

3.1.3 APARELL

(Qomariyah and Kazakov, 2017) described a learning algorithm (inspired by Progol/Aleph) called APARELL (Active Pairwise Relation Learner), APARELL is a specialized (DL-based) ILP learner for learning user preferences as DL hypotheses expressed as pairwise comparisons (e.g. $X\ betterthan\ Y$). The DL knowledge base (in the $\mathcal{EL}$ language) is represented as an OWL ontology. In APARELL, the DL hypothesis is a single DL role (object property in OWL) that compare two simple or complex concepts in the form $conceptA\ role\ conceptB$. The hypothesis language is restricted to one constant $role$, while $conceptA$ and $conceptB$ are iteratively specialized in order to reduce the coverage for negative examples. ILP learning in APARELL can be demonstrated using its car example. In the car example, the $betterthan$ role is used to learn which car preference is better than the other for a particular user. APARELL will first generate the hypothesis $(\top, \top)$: $betterthan$ (or $Thing\ betterthan\ Thing$). After that, this generated hypothesis is further specialized into $Car\ betterthan\ Car$ and then into $Manual\ betterthan\ MediumCar$ (among other generated specializations). Sometimes, a hypothesis is specialized by conjunctively adding concepts such as $Manual\ betterthan\ (MediumCar \sqcap Manual)$. By iteratively applying specializations either through replacing general concepts with more specialized ones (e.g. $Thing$ to Car), or by adding concepts conjunctively to either or both sides of the comparison; the resulting final hypothesis is a pairwise

comparison between two (possibly conjunctions of) concepts that describe ordinal preferences for a particular user.

Both of DL-FOIL and YinYang follow a similar learning strategy (i.e. the use of partial generalizations to construct hypotheses). Moreover, both are general-purpose ILP learners in DL, unlike APARELL, which is a specialized ILP learner in DL for recommender systems. Another example of a general purpose ILP learner in DL is the DL-Learner (Lehmann, 2010). The DL-Learner is commonly known as the state of the art for ILP learning in DL since it employs a set of approaches to optimize the ILP learning process in combination with its novel refinement operator and its search algorithm, which is capable of learning a hypothesis in more expressive DL languages than the three aforementioned learners. In the next sections, we review the DL-Learner in more detail since it is the foundation of this work.

3.2 The DL-Learner (a framework for DL inductive learning)

The DL-Learner was designed to learn OWL class expressions. It has its own set of learning algorithms specifically designed for DL-based ILP learning, each of which targets specific learning applications and DL languages. Although the DL-Learner has been extended to include more capabilities recently, it has become a framework for ILP learning in DL (Bühmann, Lehmann and Westphal, 2016). The DL-Learner is considered the state of the art in DL ILP learning because unlike other DL-Learners, such as DL-FOIL (Fanizzi, d'Amato and Esposito, 2008), the DL-Learner heavily employs information from the axioms in the TBox and statistics about the ABox assertions to prune the search space and to guide the hypothesis search more intelligently. In addition, the DL-Learner checks the redundancy of a concept (hypothesis) and its weakly-equal variations against the search tree using an efficient procedure, which prunes unnecessary areas of the search space, therefore improving the learning performance even more. Given the DL-Learner's capabilities, it would be a strong candidate for building an efficient and scalable ILP learner in DL. An overview the DL-Learner framework can be seen in Figure 3.1.

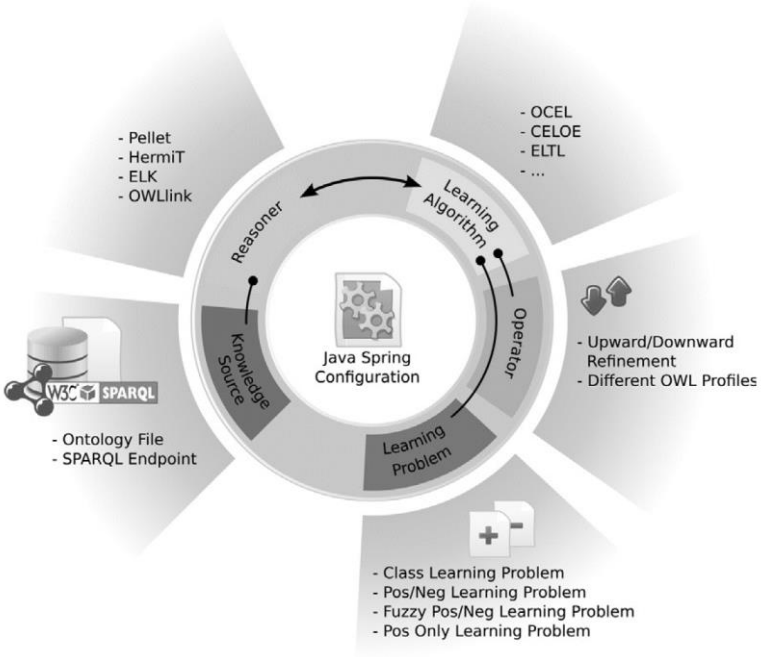


Figure 3.1: An overview of the DL-Learner framework (Bühmann, Lehmann and Westphal, 2016).

The DL-Learner framework consists of five components. First, the Knowledge Source component handles the loading of input data from either a file or a result of a SPARQL query. The second component is the Reasoner, which supports a list of DL reasoners for reasoning purposes, especially for TBox materialization. The third component is the Learning Algorithm, which has a list of the DL-Learner-supported learning algorithms, each of which targets learning in a specific DL language, such as ELTL (EL Tree Learner) for the $\mathcal{EL}$ language. In addition, the DL-Learner framework has the CELOE (Class Expression Learner for Ontology Engineering) algorithm which specifically targets ontology engineering. Moreover, the DL-Learner has an algorithm for general purpose learning in DL, known as the OCEL algorithm. The fourth component is the Operator, which supports a set of refinement operators (e.g. upward, downward). The fifth component is the Learning Problem, which handles the configuration of the learning process (e.g. only using positive examples for learning). In the next section, we review the OCEL algorithm (Lehmann, 2010) in detail.

3.3 The OCEL (OWL Class Expression Learner) algorithm

The OCEL algorithm (OWL Class Expression Learner) uses a top-down-informed tree search to learn hypotheses expressed in the DL language $\mathcal{ALCQH}^{(\mathcal{D})}$, and it is suitable for many real-world knowledge bases. As part of the OCEL algorithm, the subsections below introduce its refinement operators, scoring function, and search procedure.

3.3.1 Refinement operators

The OCEL algorithm targets the DL language $\mathcal{ALCH}$ as its baseline through its refinement operator. Its baseline refinement operator can, however, be extended to learn more expressive DL hypotheses

(e.g. expressed in the $\mathcal{ALCQH}^{(\mathcal{D})}$ language). See Table 3.2 for a comparison of the two variations of OCEL refinement operators (p).

Table 3.2: A comparison of the DL-Learner's refinement operators for OCEL.

The DL-language (of the refinement operator)	**Complete**	**Proper**	**Finite**	**Redundant**
$\mathcal{ALCH}$	Yes (weakly)	No	No	Yes
$\mathcal{ALCQH}^{(\mathcal{D})}$	No	No	No	Yes

Both of the OCEL refinement operators are improper, infinite, and redundant. However, according to a detailed analysis of these refinement operators (Lehmann, 2010), they perform effectively in real-world learning. In term of completeness, the baseline operator for $\mathcal{ALCH}$ is weakly complete, which means that for any concept $C \sqsubseteq \top$, there exists a refinement path (i.e. a sequence of refinement steps) from $\top$, with which a concept E (where $E \equiv C$) can be reached. For properness, the refinement operator is improper, which means that the refinement of a concept C may generate a refinement D, where $D \equiv C$. The refinement operator being infinite means that the refinement of a concept C does not terminate unless limited by some measure (e.g. hypothesis length). For redundancy, the refinement operator being redundant means that there exist multiple refinement paths, with which the refinement D can be reached from C. In addition, when refining a concept C in the same refinement path from a concept D, a weakly-equal concept to one of C refinements is generated. For an example of weakly equivalent concepts (hypotheses) see Figure 3.2.

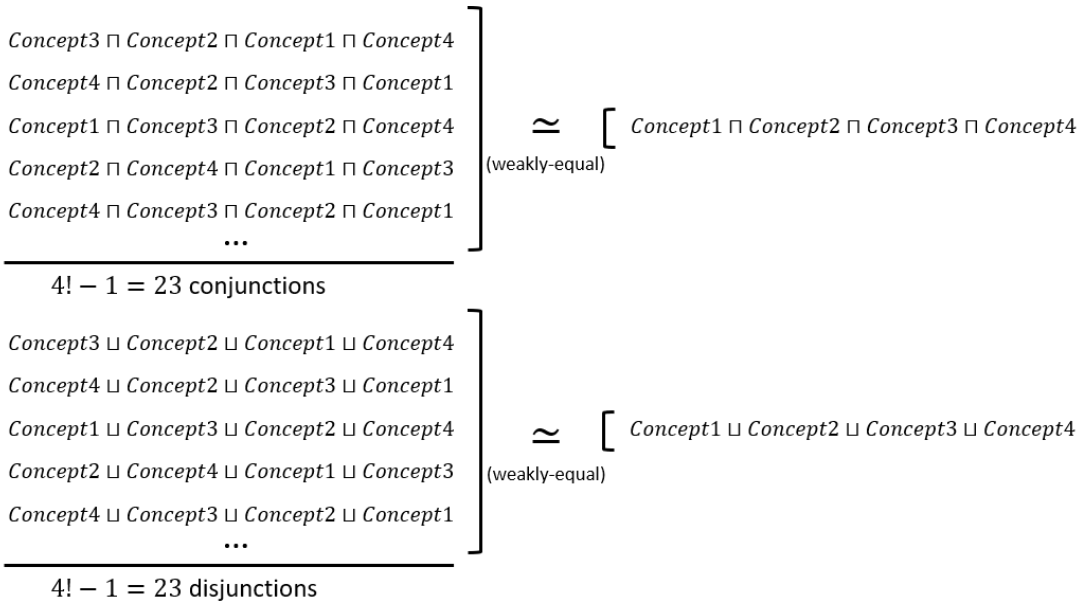


Figure 3.2: An example of weakly-equivalent concepts.

In Figure 3.2, the hypotheses that share the same operands, regardless of their order and the same operation (conjunction/disjunction), are said to be weakly-equal.

The second refinement operator for $\mathcal{ALCQH}^{(\mathcal{D})}$ is similar to its baseline operator $\mathcal{ALCH}$. However, since the second refinement operator is extended to support cardinality restrictions on roles and concrete roles with float and Boolean values, it becomes incomplete; this designation is due to the continuous range of float values for concrete roles.

Both of the refinement operators incorporate knowledge from the knowledge bases TBox and its RBox. The RBox provides the role axioms with which a role domain and range is determined. In addition, the RBox is used to infer the role hierarchy using its role inclusion axioms; this inference is important when refining roles; for example, considering the RBox axiom $hasSon \sqsubseteq hasChild$, the refinement operator may refine the concept $\exists hasChild.\top$ into $\exists hasSon.\top$ to only include male children.

The refinement operator uses the class hierarchy from TBox as well as the role hierarchy and domain range restrictions from RBox to generate better downward refinements that navigate the DL hypothesis space more intelligently. When the refinement operator for $\mathcal{ALCQH}^{(\mathcal{D})}$ is used, additional information about the roles and concrete roles are incorporated. For roles, the maximum number of role fillers per role (i.e. the maximum number of times that an individual is used as a subject for specific role assertions) is collected. The statistics about role fillers are used to generate better cardinality restriction refinements. For concrete roles (i.e. Boolean and float), the values for a given float concrete role are discretized (based on their values) into splits (list of float values); the number of splits (or values) is user-defined. When generating refinements for float concrete roles, cardinality restriction values are extracted from the splits. In other words, the split values of each float concrete role are only used for its refinements. For the Boolean concrete roles, two refinements per concrete role are generated: the True value and the False value.

In terms of refinements, disjunctions are only generated as part of refining the top concept. The refinements of the top concept are also used when refining other concepts – for example, by replacing a part of a concept being refined or by adding it conjunctively. Because of the infiniteness of the two refinement operators, the refinements are generated up to a given (max) hypothesis length, which is then gradually increased (by OCEL). An OWL ontology about Michalski's trains ("trains.owl")[14] demonstrates the OCEL refinement operators in action; the refinements of the top concept (Thing) with length = 2 on the baseline $\mathcal{ALCH}$ operator can be seen in Figure 3.3.

[14] Available as one of DL-Learner's example datasets at: https://github.com/SmartDataAnalytics/DL-Learner/releases/tag/1.4.0

<table>
<tr><td colspan="2">Michalski’s trains OWL ontology

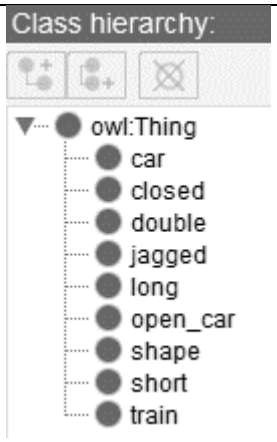
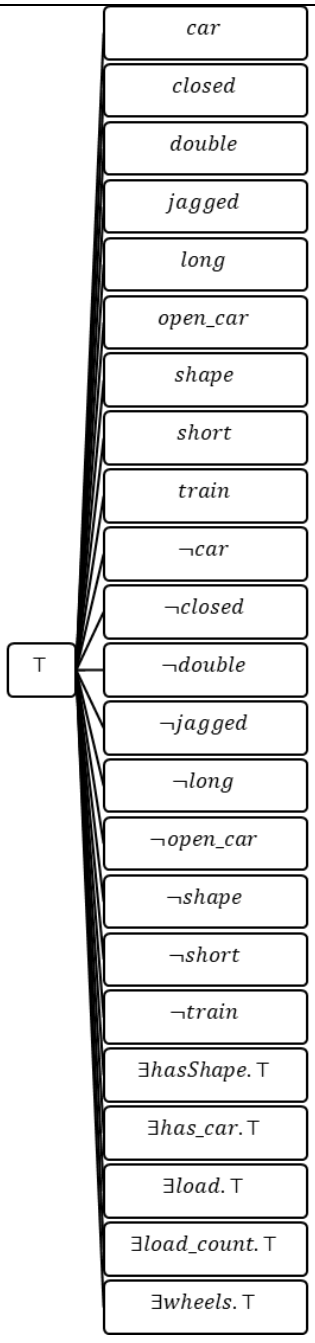
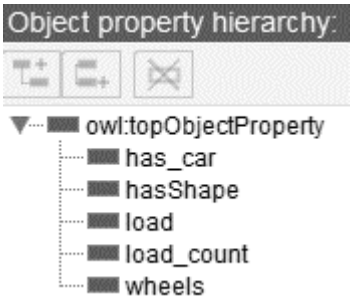
(in Protégé)</td><td>The refinement of the top concept ⊤ with length=2</td></tr>
<tr><td>Classes:</td><td> 
</td><td rowspan="2"> 
</td></tr>
<tr><td>Roles:</td><td> 
</td></tr>
</table>

Figure 3.3: Refinements of the top concept of Michalski’s trains.

In Figure 3.3, the refinements (hypotheses) of the top concept are the union set of

- Atomic classes
- Negation of atomic classes
- Existential role restriction of atomic roles on the top concept “Thing”
- Universal role restriction of atomic roles on the top concept “Thing”

- Disjunction of concepts

A hypothesis in the above union set, where its length is ≤ 2, is then returned as a refinement of the top concept with a length of 2. When refining the same top concept with a length of 3, three additional types of refinements are generated – namely, the set of universal role restrictions on atomic roles (e.g. $\forall has\_car(Thing)$), the set of disjunctions and the set of conjunctions between two atomic classes are generated (e.g. $long \sqcup short, closed \sqcap short$). When refining a concept C with length n, the resulting refinements include all refinements in the length $\leq n$. A hypothesis length is the sum of sub-hypotheses' lengths, where the length of each sub hypothesis is determined by its DL operator type (conjunction, disjunction, universal restriction, etc.) and by the number of its operands. For example, the universal restriction $\forall r.C$ is of length 3, which is longer than $\exists r.C$, which is in length 2; both are identical in their actual length (i.e. a single role restriction on a simple concept). The DL-Learner employs this approach of computing length in order to prioritize which DL operator (s) are used first to construct hypotheses (i.e. a hypothesis that uses simpler DL operators to explain the learning examples is preferred over a hypothesis that uses more complex ones. An example of hypotheses in a particular length is shown in Table 3.3.

Table 3.3: An example of a hypotheses in specific lengths.

Hypothesis length	**Allowed hypothesis**
Length 1	• Simple concept
Length 2	• Negated simple concept • Existential restriction on a simple concept • Concrete role cardinality restriction
Length 3	• Universal restriction on a simple concept • Disjunction between two simple concepts • Conjunction between two simple concepts
Length 4	• Role cardinality restriction on a simple concept
Length n	• Disjunction between two complex concepts where the disjunction length + the length of its two complex concept operands is $\leq n$ • Conjunction between two complex concepts where the conjunction length + the length of its two complex concept operands is $\leq n$ • Role restrictions on complex concepts with total length $\leq n$

The OCEL algorithm keeps only unique search nodes in the tree via the use of redundancy checks (which also handle the redundancy of its refinement operators). The redundancy check in OCEL checks whether the search tree already contains a weakly equivalent hypothesis. The redundancy check technique enforces a deterministic sorting of operands. Consequently, the size of the search tree (and its branches) is dramatically reduced (or pruned); therefore, a faster (more efficient) DL learning is achieved.

3.3.2 The scoring function

The OCEL algorithm combines multiple factors when evaluating a node N. A search node $N = (C, n, b)$ contains the concept (hypothesis) C, its upper limit for child hypothesis length (i.e. the horizontal expansion) n, and a flag variable b to indicate redundancy (true is redundant; otherwise, it is false). According to equations 3.1 and 3.2, as reproduced from (Lehmann, 2010, p. 92), a search node is evaluated as follows. First, the accuracy of the node (hypothesis) is calculated using equation (3.1) by subtracting from 1 the division of the sum of uncovered positives up plus the covered negatives cn divided by the total number of training examples $|E|$. Once the accuracy is calculated, it is then used to calculate the accuracy gain, as in equation (3.2), which is the accuracy of the current hypothesis minus the accuracy of its parent.

$$accuracy(C) = 1 - \frac{up + cn}{|E|} \tag{3.1}$$

$$acc\_gain(N) = accuracy(C) - accuracy(C') \tag{3.2}$$

where C' is the concept (hypothesis) of its the parent (search) node

Before calculating the accuracy and its gain, the value of up is first checked to determine whether it is within the acceptable noise tolerance levels against the formula ($up > \lfloor noise \cdot |E| \rfloor$); if up exceeds the acceptable noise levels, it is considered too bad (noisy) and is not expanded. The noise tolerance is user-defined as a percentage between 0% and 100%, where 0% indicates learning from noise-free data. If candidate hypothesis (refinement) is within the acceptable noise range, its accuracy and accuracy gain are calculated. In addition, its score is calculated as follows:

$$score(N) = accuracy(C) + \alpha \cdot acc\_gain(N) - \beta \cdot n \quad (\alpha \geq 0, \beta > 0) \tag{3.3}$$

The score combines three criteria. The first (main) one is the predictive accuracy. The second criterion is the accuracy gain, which is tuned by α to favour hypotheses that lead to more accurate ones (i.e. increase in accuracy). The third criterion tunes the search bias (through β on the node's horizontal expansion n) to favour shorter hypotheses. In the OCEL algorithm, the values for α and β are set to 0.5 and 0.02, respectively, by default. An example for computing the OCEL score can be seen in Figure **3.4**.

Search nodes info

Learning configuration: 8 positives, 5 negatives, and 0% noise tolerance

Node name	Covered positives	Covered negatives	Accuracy Using equation (3.1)
parentNode	8	5	$accuracy = 1 - \frac{(8-8)+5}{(8+5)} = 0.615 = 61.5\%$
childNode1	8	0	$accuracy = 1 - \frac{(8-8)+0}{(8+5)} = 1 = 100\%$

The search tree

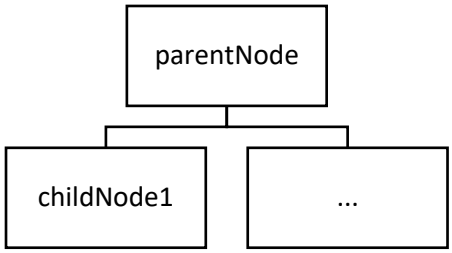


Figure 3.4**: An example of computing the OCEL score.**

Figure 3.4 assumes that *childNode1* has a horizontal expansion $n = 3$. After computing the accuracy of *childNode1*, the accuracy gain is calculated using equation (3.2) as $acc\_gain = 1 - 0.615 = 0.385 = 38.5\%$. Once the accuracy gain is computed, the score is computed using equation (3.3) as follows:

$$score = accuracy + \alpha \cdot acc\_gain - \beta \cdot n = 1 + 0.5 \cdot 0.385 - 0.02 \cdot 3 = 1.1325$$

3.3.3 Inductive learning in OCEL

The OCEL uses a top-down-informed tree search. The OCEL algorithm handles the redundancy of its refinement operator through redundancy checks (as discussed in section 3.3.1 above) and its infiniteness through gradual generation of refinements (up to a given hypothesis length he, known as the horizontal expansion). OCEL may expand the same (search) node multiple times; however, the value for he is increased each time, therefore, more child hypotheses are allowed to be generated. See Algorithm 3.1 for the pseudocode for the OCEL algorithm.

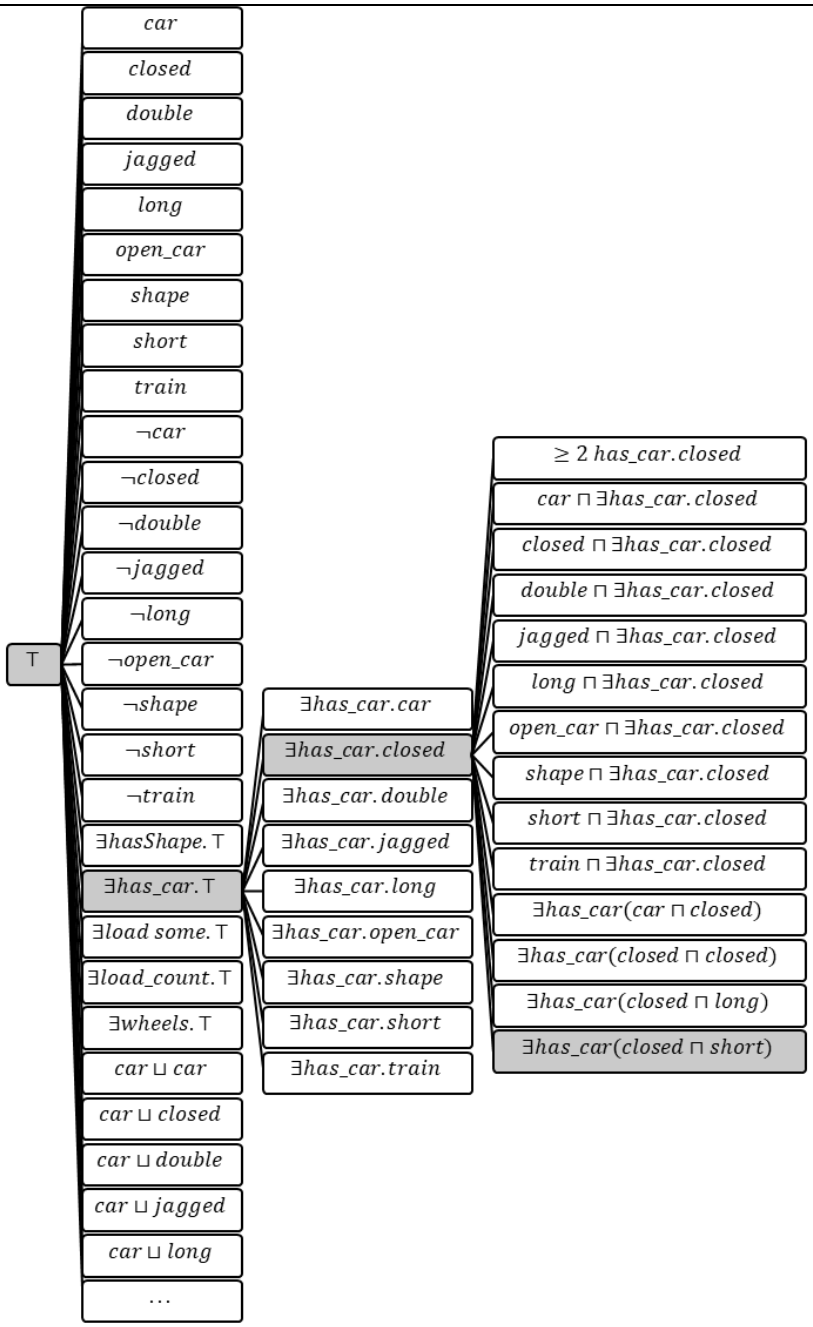
Algorithm 3.1: OCEL (OWL Class Expression Learner) algorithm (Lehmann, 2010, p. 93).

```
Input: background knowledge, examples E, noise in [0,1]
1 ST (search tree) is set to the tree consisting only of the root node
  (T,0,false)
2 while ST does not contain a node with q < |noise · |E|| do
3   choose a node N = (C,n,b) with highest score in ST
4   expand N up to length n+1:
5   begin
6     add all nodes (D,n,checkRed(ST,D)) with D ∈ transform(p^cl(C))
      and |D| = n+1 as children of N
7     evaluate created non-redundant nodes
8     change N to (C,n+1,b)
9   end
10 Return found concepts in ST
```

Algorithm 3.1 takes the following inputs: the background knowledge, positive and negative training examples E, and the noise tolerance percentage with a value between 0 and 1. A search node is represented as $N = (C, n, b)$, where C is the hypothesis, n is the maximum hypothesis length for child refinements, and b is a flag variable to indicate a hypothesis redundancy in the search tree. Initially, the search tree is set to only contain the top concept in line 1, with $n = 0$. When any search node is expanded, it is expanded to $n + 1$, which means that a generated child hypothesis is equal to or longer than its parent. The same search node may be expanded multiple times; however, its n is different each time; any increase to n for a given hypothesis, results in more child hypothesis being generated since n limits the maximum hypothesis length for child hypotheses.

The algorithm starts with a loop iteration in line 2. In that loop iteration, a node $N = (C, n, b)$ that has the highest score in the search tree (line 3) is expanded (or re-expanded) up to its $n + 1$ (line 4). For every generated child node, the (child) hypothesis is transformed into ordered negation normal form D, and then its redundancy is checked against the search tree to see if a weakly-equal concept already exists; the result of the redundancy check (true or false) is stored in the child's node flag variable b (line 6). After all the newly generated child nodes are redundancy checked, the scores for non-redundant nodes are calculated in line 7. Thereafter, the parent node's upper expansion limit (n) is increased by 1 (line 8). The learning continues through the loop iterations until a termination criterion is met (e.g. no search tree node is worth expanding); in practice, a timeout is typically used. In order to demonstrate the algorithm in action, an example is shown in Figure 3.5 for learning about Michalski's eastbound trains on the same dataset used in Figure 3.3.

Search tree



OCEL learning iterations

Iteration action	Resulting new refinements
1. Expand "Thing" with $n = 1$	• Set of atomic concepts
2. Re-expand "Thing" with $n = 2$	• Set of negated atomic concepts • Set of existential restriction on each role with the top concept ⊤

3. Re-expand "Thing" with $n = 3$	• Set of universal restriction on each role with the top concept ($\top$) • Set of disjunctions between two atomic classes
4. Re-expand "Thing" with $n = 4$	• Set of disjunctions on two atomic classes with one being negated • Set of disjunctions on an atomic class with one existential restriction on the top concept • Set of MAX cardinality restriction on each role with its corresponding max role filler on the top concept $\top$
5. Re-expand "Thing" with $n = 5$	• Combination of disjunctions between the above refinements where the length of each disjunction is ≤ 5
6. Expand "$\exists has\_car.\top$" with $n = 3$	• Set of existential restriction on the "has_car" role on each atomic concept
7. Re-expand "$\exists has\_car$.closed" with $n = 3$	• No new refinements generated
8. Re-expand "$\exists has\_car$.closed" with $n = 4$	• the following MIN role cardinality is generated: "≥ 2 has_car.closed" • Set of conjunctions of each atomic concept with "$\exists has\_car$.closed" e.g. "car $\sqcap \exists has\_car$.closed" • Set of conjunctions on the role restriction's concept with each atomic concept e.g. "$\exists has\_car$(closed $\sqcap$ long)"
Solution found at iteration 9: "$\exists has\_car$(closed $\sqcap$ short)" with 100% accuracy.	

Figure 3.5: An example of learning the description of Michalski's eastbound trains.

In Figure 3.3, the OCEL algorithm is assigned to learn the description of the trains heading east. The positive examples are the five trains heading east (eastbound), and the negative examples are five trains heading west (westbound). The effect of increasing the expansion limit n when re-expanding a hypothesis is clearly observed in every iteration, through which new types of refinements are generated for the same node. The algorithm learned the description of eastbound trains in nine iterations with the following solution path:

$$\top \rightarrow \exists \text{has\_car}.\top \rightarrow \exists \text{has\_car}.\text{closed} \rightarrow \exists \text{has\_car}(\text{closed} \sqcap \text{short})$$

The OCEL algorithm is a very capable learning algorithm, and it incorporates varied information and statistics about the knowledge base through its refinement operators; the result is a learner that navigates

the DL search space more intelligently and efficiently. However, even though the OCEL algorithm is capable of learning from real-world noisy datasets, it still suffers from the same scalability issues as many ILP learners (scalability issues for ILP learners in general are discussed in chapter 4). Addressing the scalability issues for a capable learner in DL, such as the DL-Learner, is a great step towards expanding ILP learning in DL into a large-scale machine learning technique.

3.4 Summary

The main purpose of chapter 3 is to review ILP learning in the context of DL (as its knowledge and model representation). Throughout this chapter, the nature of ILP learning in DL is discussed, and related work in DL-based ILP is assessed. As a result of this chapter's analysis, it is clear that the DL-based inductive learner (the DL-Learner) is the state of the art and has unique strengths in the following areas:

- Construct more appropriate refinements for expressive languages (e.g. $\mathcal{ALCQH}^{(\mathcal{D})}$) by using information from the background knowledge:
    - Class and role hierarchies.
    - Roles and concrete roles statistics.
    - Concrete roles discretization.
- Prune unnecessary areas of the search space:
    - Reducing all possible combinations of operands' order for a conjunction or disjunction into a single combination.
    - Avoiding the expansion of already bad hypotheses.

The technical details of the DL-Learner are closely examined, including a discussion of its general-purpose inductive learning algorithm (OCEL), its refinement operators, and its scoring function.

Although the DL-Learner is powerful in learning expressive DL hypotheses efficiently, it is still a sequential ILP learner and therefore suffers from the same scalability issues as many ILP learners.

In the next chapter, we explore and analyse the parallel strategies and approaches for accelerating ILP- and DL-related computations.

Chapter 4: Parallelization strategies

In the previous chapters, we review fundamental concepts in ILP, DL, and parallel computing. We also look closely at the state of ILP learning in DL, the DL-Learner. In this chapter, we review the existing techniques and approaches to accelerate ILP- and DL-related computations, with a particular focus on the use of parallel computing and parallelization strategies. These parallelization strategies for ILP and DL are discussed separately following a review of the non-parallel strategies. The structure of the review is highlighted in bold in Figure 4.1. It is worth noting that in this thesis, the focus is mainly on accelerating ILP computations using parallel computing. The application of non-parallel approaches such as Query Packs (a form of caching) and sampling techniques, are not considered (in this thesis), even though they will provide additional performance gains; the focus on these non-parallel approaches will remain as part of potential future work. However, some non-parallel approaches will be used as a consequence of basing this thesis on the DL-Learner (the state-of-the-art in DL-based ILP learning), which has its own non-parallel approaches.

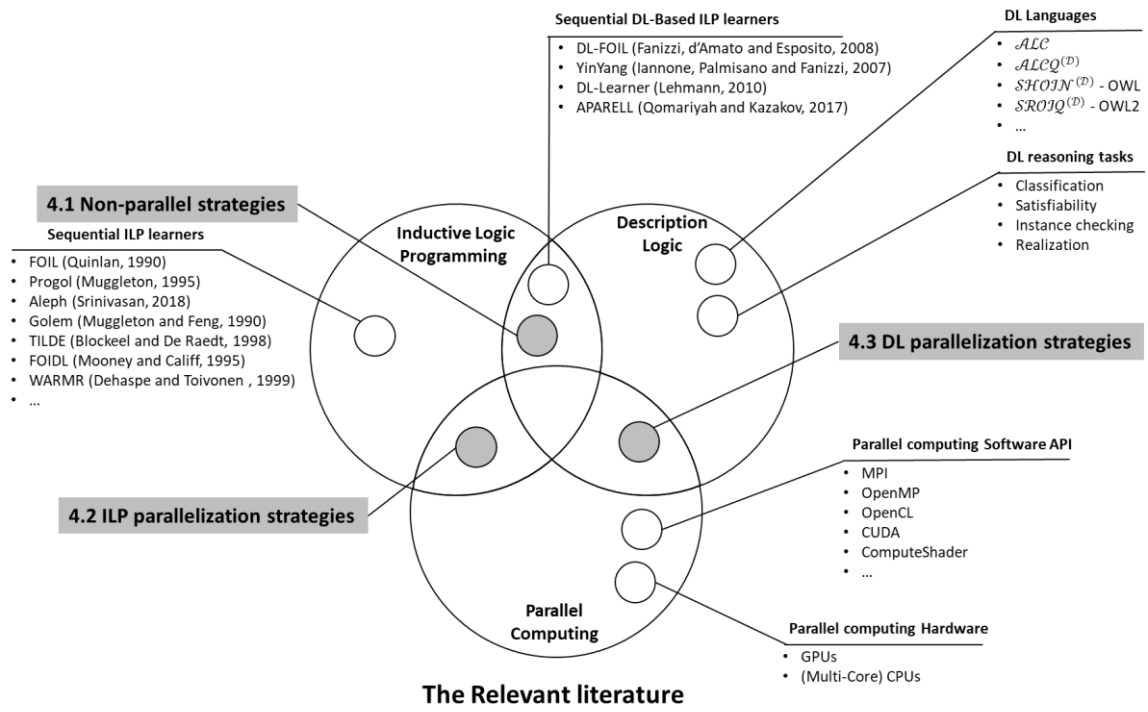


Figure 4.1: The structure of the review.

4.1 Non-parallel strategies

In this section, we review the non-parallel acceleration strategies to improve ILP learning. The ILP learning process consists of three procedures: hypothesis search, hypothesis generation, and hypothesis evaluation; accelerating one or more of these procedures improves the learning performance. The performance of hypothesis search depends largely on the hypothesis generation since hypothesis generation controls the size and complexity of the hypothesis space. In order to improve the

performance of the hypothesis search, knowledge about the learning problem is used to constrain the hypothesis space into more suitable hypotheses; this constraining can be either be automatically identified from the learning dataset by the ILP learner or known from a domain expert. Sometimes, classic search optimization approaches are used to improve search algorithms, such as avoiding or pruning search paths that lead to worse hypotheses. In the case of ILP learning in DL, the appropriate selection of the DL language can dramatically reduce the hypothesis space into a smaller and more manageable one; a simpler DL language means less computation needs. Moreover, a simple yet useful way to improve ILP learning is to impose a maximum hypothesis limit.

There are ILP-specific approaches to improve the ILP learning process, such as query packs (Blockeel *et al.*, 2000). The authors proposed a method for grouping and executing similar queries, where intermediate results can be reused among them; this method effectively reduces redundant computations. They call their technique "query packs". A query pack is a set of conjunctive queries represented as a tree structure. A query pack may contain multiple child (or sub) query packs. The authors developed a Prolog engine that supports query packs natively, known as ILProlog, where they tested their approach by reimplementing the two ILP learners with query packs: TILDE (Blockeel and de Raedt, 1998) and WARMR (Dehaspe and Toivonen, 1999). The experimental results suggest that the larger the query pack is, the more performance improvement is achieved.

Another ILP-specific approach to improve performance is to use sampling techniques (Srinivasan, 1999) to reduce the size of the learning examples to smaller, yet representative, examples to reduce the time of hypothesis evaluation. The researcher proposed two subsampling techniques, subsampling and logical windowing. For subsampling, a user defined limit m determines the size of a subsample; if the number of learning examples exceeds m, a subsample is taken at random to compute the hypothesis score. For logical windowing, a hypothesis is constructed from a sample (window) from the learning examples. This hypothesis is evaluated on examples outside the window, and if it returns sufficient classification errors, the window is then expanded by adding more examples. Once the window is expanded, a new hypothesis is constructed. The window is gradually expanded during learning. In comparison with subsampling, logical windowing is more sophisticated since it adapts to the learning problem by using more examples when necessary.

The aforementioned approaches improve the ILP performance – especially when combined. Even though these approaches address the ILP scalability issue, they do not do so sufficiently. On the other hand, since parallel computing improves performance by dedicating the computing power of several processors towards a single problem, it is potentially useful for addressing the scalability of ILP learning. In the next section, we review the parallelization approaches used to accelerate ILP- and DL-related computation.

4.2 ILP parallelization strategies

In this section, we review the parallelization approaches to accelerate ILP hypothesis search, evaluation, and their combination (i.e. parallel search and parallel evaluation), but we first consider the existing classifications of parallel ILP systems in the literature. Fonseca *et al.* (2009) surveyed and classified parallel ILP systems into three categories: search, data, and evaluation. The search category covers the ILP systems that conduct multiple searches in parallel or that parallelize a single search. For the ILP systems in the data category, the data is divided among processors before the learning starts, and each processor does a specific task to its portion of data; it may do so independently or collaboratively with other processors. For example, each processor may build a weak hypothesis based on the data available to it; these weak hypotheses from multiple processors are then aggregated into a strong final hypothesis. For ILP systems in the evaluation category, the evaluation of a candidate hypothesis is parallelized. As pointed out by the authors, the aforementioned categories (strategies) can be used together. Many of the ILPs that employ the search strategy have shared memory architecture, whereas the parallel data and evaluation strategies are existent in both the shared and distributed memory architectures. The authors also observe that data parallel approaches always achieve speedups and that when the training data is complex and/or large enough, significant speedups can be attained with the parallel evaluation strategy.

4.2.1 Parallel hypothesis search

Nishiyama and Ohwada (2017) proposed a parallel ILP system, the architecture of which can be seen in Figure 4.2. The system has four components: server, master, worker, and ILP modules. The server module acts as the communication point between the worker modules. The master module can communicate with the worker modules directly or through the server module (e.g. message broadcast). The master module sends the learning task to one of the worker modules, which in turn feeds the data to the ILP module. The ILP module is simply a classic ILP learner that uses inverse entailment controlled by the worker module. There are three communication protocols for the communication among the modules. The first is the protocol whereby worker modules communicate with each other through the server module; this communication is done by searching for an available worker module. The second protocol is for the direct communication between the worker modules. In other words, it is a peer-to-peer communication. When the workload of a worker module reaches a certain limit, it communicates with other workers to balance the workload. When a worker module completes the assigned work, it starts to work on the task sent from other workers; the key is to keep all workers busy all the time to achieve efficiency. The master module simply initiates the learning process and receives the results back; the division of workload is done by the workers continuously keeping each other working by balancing the workload among them. The worker module can communicate with other workers while its corresponding ILP module is learning – therefore using time more efficiently. The authors have tested their approach with 20 CPUs (two machines with 6 CPUs each and two machines

with 4 CPUs each), and they achieved a speedup 15.8 times greater than the baseline. The speedup kept increasing with the number of CPUs.

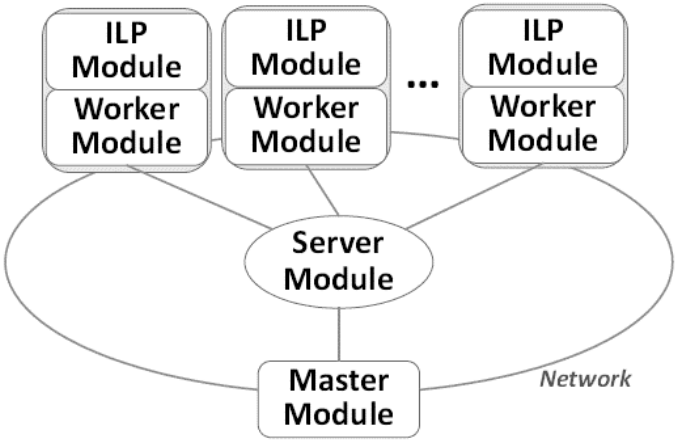


Figure 4.2: Architecture of the parallel ILP system (Nishiyama and Ohwada, 2015).

In one study (Ohwada, Nishiyama and Mizoguchi, 2000), a distributed parallel search algorithm was proposed for inverse entailment (such as in Progol). The researchers formalized Progol search into a branch-and-bound search, where the branches of the search are explored in parallel by distributed processors. Their approach works as follows. First, a Progol-like algorithm is implemented, which works exactly like the traditional Progol; however, the hypothesis space is explored in parallel. In this distributed parallel exploration (searching), the search is broken down into multiple smaller searches, which are then distributed to other processors. The communication between the processors is done through a simplified adaptation of the Contract Net Protocol (CNP) that works in three steps. First, a processor checks for an available processor by broadcasting a message that is received by all other processors. The available processors reply to the broadcasting processor about their availability. Thereafter, the broadcasting processor selects one of the available processors and then assigns a search task to it. There are two ways to receive the output from a processor: with a supervisor and without. In the supervised way, the results of each search by the other processors are sent to the supervisor processor, whereas in the non-supervised way, the result is reported to the calling processor, which initiated or assigned the task. Based on the authors' experiments, significant speedups were achieved, more so with the supervised way and less with the unsupervised approach.

4.2.2 Parallel hypothesis evaluation

Zeng, Patel and Page (2014) proposed an algorithm called QuickFOIL, inspired by the classic algorithm FOIL. QuickFOIL can cope with a large amount of data with custom evaluation function and pruning strategy. For the custom evaluation function, a custom function that combines the Matthews correlation coefficient (MCC), area under the entropy (AUE), and F_β measure is used to evaluate the generated candidates. In this function, MCC is used to evaluate the performance of binary classifiers, and it takes

into account TP (True Positive), FP (False Positive), TN (True negative), and FN (False Negative). The QuickFoil evaluation function is calculated using Equation 4.1.

$$f(L) = \frac{1 + \beta^2}{\beta^2 . \frac{1}{MCC + 1} + \frac{1}{AUE(p') - AUE(P) + 1}}$$

Equation 4.1: QuickFOIL evaluation function (Zeng, Patel and Page, 2014).

The parameter β (set to 2 as the default) is used to tune the weight of MCC and AUE on the final equation. In QuickFOIL, a pruning strategy is employed in order to limit the search. In the pruning strategy, redundant literals are pruned through what the authors call "replaceable duplicates", which are defined by the authors as "A literal L is a replaceable duplicate with respect to a clause C if there is a literal L' in the body of the clause such that the new clause that replaces L' in C with L is equivalent to C" (Zeng, Patel and Page, 2014, p. 5). The QuickFOIL algorithm can utilize relational databases (RDBMS) for data storage and query answering, therefore achieving scalability for ILP. In relation to RDBMS, a predicate map into a database table and a hypothesis map into a SQL query evaluated against these tables (predicates). The SQL "COUNT" queries are used to obtain the values for the parameters necessary to calculate the result of the scoring function for every candidate hypothesis. When QuickFOIL was tested with several datasets in the RDBMS environment, it outperformed classic FOIL and Aleph in terms of execution times and F-score performance measure. QuickFOIL also discovered a shorter list of rules than other algorithms (e.g. FOIL and Aleph).

Another researcher (Konstantopoulos, 2007) parallelized the evaluation of hypotheses for Aleph. The approach works by dividing the data among the processors. The topology of the communication between the processors is having the master processor that partitions the data and then distribute it to other processors (workers) and then having every processor process a subset of the data. The communication between the processors is done through MPI. Partitioning the data into multiple smaller partitions reduces the memory and computation requirements needed for a single processor to process larger data that may not fit in the memory. The proposed approach works as follows. First, the master node (processor) partitions and distributes the data to the workers. Second, the master runs sequential Aleph; however, when it reaches the clause evaluation, it sends the clause to other processors (workers); each worker then evaluates the clause based on its subset of the data. Thereafter, each worker sends the covered examples and their numbers to the master, which then calculates the evaluation score for the clause by using numbers collected from all workers. An overview of data parallel Aleph can be seen from Figure 4.3.

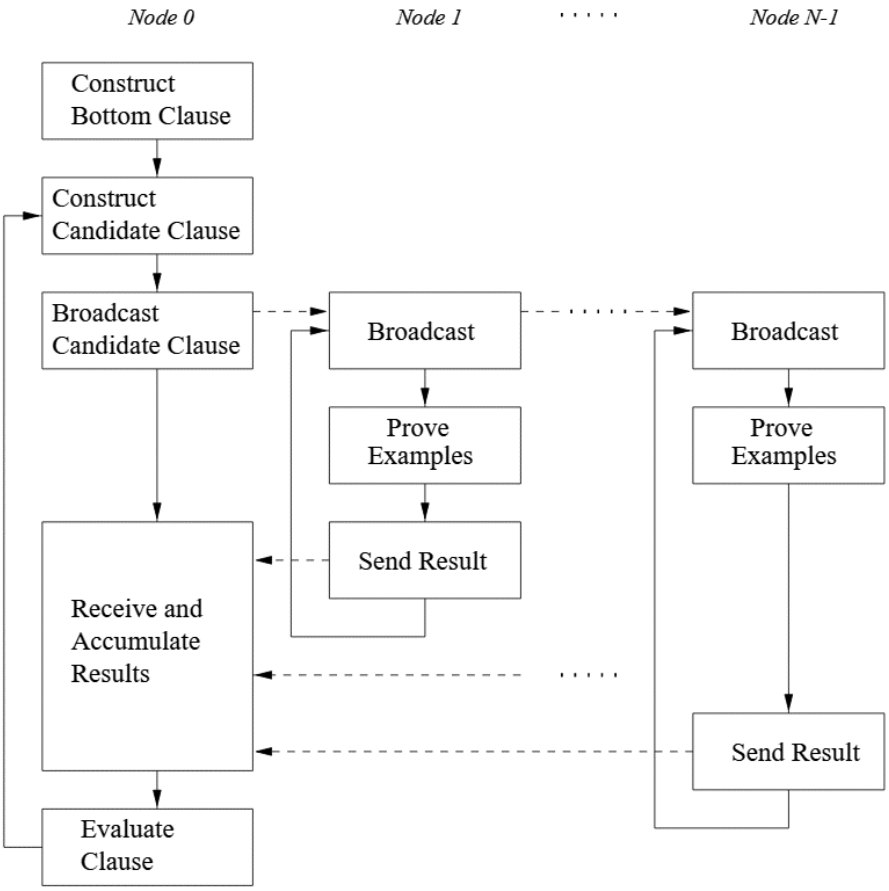


Figure 4.3: An overview of data parallel Aleph (Konstantopoulos, 2007).

Another study (Srinivasan, Faruquie and Joshi, 2010) proposed the use of MapReduce paradigm for conducting the coverage test for the Aleph system. This approach consists of three sequential steps that run in parallel. In the first step, a function is invoked that takes the clauses and the examples; the result of the function is the Cartesian product of clauses $\times$ examples. Thereafter, a map function that takes the previous results formatted as (clause, example) and produces a list of Key-Values pairs, where the Key is the clause and the Value is the confusion matrix for that example. In the Reduce steps, the Key-Value pairs with the same key are grouped together, and the total confusion matrix for that clause is calculated; the output is in the form of (clause, total confusion matrix). The final step is to use the total confusion matrix to calculate the cost function; the output would be in the form of (clause, cost function). The authors conducted experiments with both synthetic data (5 GB) and real-world data (7GB). They observed that in order to achieve speedups, the data needs to be massive enough because smaller data reduces the speed; if the data is small enough, performance worse than the serial version of the algorithm is attained.

Another team of researchers (Martínez-Angeles *et al.*, 2016) proposed an ILP learner based on Aleph, which accelerates the coverage computations with GPU hardware. The coverage computations are done through the GPU-accelerated Datalog engine. Datalog is a query language that extends to relational

databases to support some logic programming semantics (e.g. recursive queries); such an extension is also known as a deductive database, where deductive reasoning can be performed on relational databases. The proposed engine follows the Datalog's bottom-up evaluation strategy (instead of top-down) as it is GPU-friendly. See Figure 4.4 for an overview of the proposed Aleph-based learner.

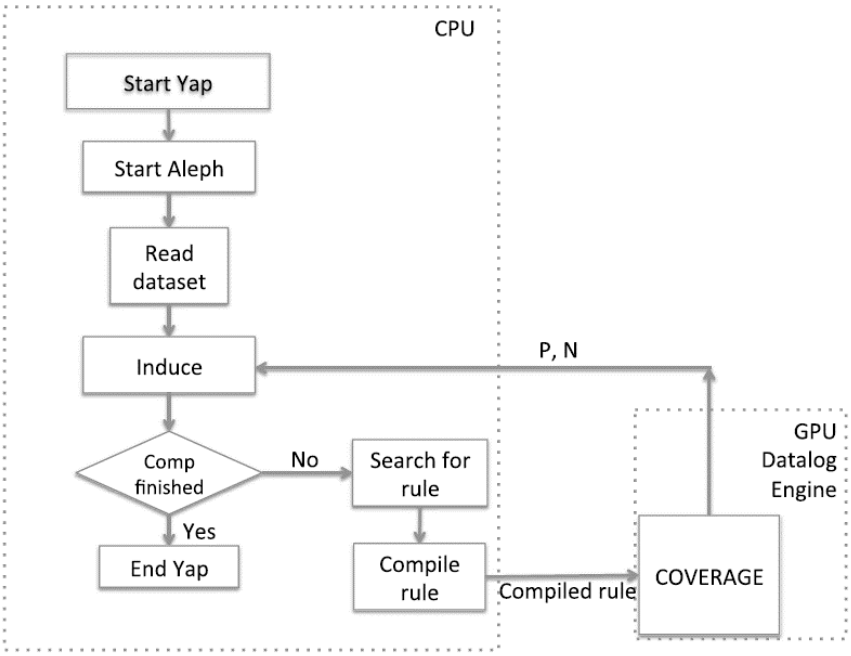


Figure 4.4: GPU-accelerated Aleph learner (Martínez-Angeles *et al.*, 2016).

The proposed learner works as follows: First, the YAP Prolog system starts. Second, the Aleph algorithm starts, and the dataset is loaded and mapped onto the GPU memory (i.e. the Datalog engine). After that, the ILP learning starts using Aleph in the 'induce' step; on only the first call of 'induce', the bottom clause is constructed. In the 'Search for rule' step, a generated rule (or candidate hypothesis) by Aleph is then converted (in 'Compile rule') into a series of GPU-based relational algebra operations, and then sent to the GPU for coverage computation. Once the GPU completes the assigned computations (in the 'COVERAGE' step), only the numbers of covered positives and negatives are returned back to Aleph so that a candidate hypothesis is assessed based on its computed coverage; a coverage list is not used, since the GPU always evaluate a hypothesis on all examples for better utilization of the GPU hardware. In the 'Comp finished' step, the termination condition for Aleph is checked (e.g. whether a rule is found that cover all positives and no negatives).

4.2.3 Parallel hypothesis search and evaluation

Ohwada and Mizoguchi (1999) suggested multiple methods that parallelize the ILP search. The methods reside on top of *KL1*, a parallel logic programming language. The methods parallelize the branch-and-bound search, into which the ILP search was formulated by the authors. In the first method (*induce*), several independent, more specialized hypotheses (originated from a more generalized parent hypothesis) are explored in parallel. In the second method (*branch*), the process of hypothesis branching

(i.e. the breaking of more general hypotheses into multiple independent, more specialized ones) is done in parallel too; this breakdown (or branching) of a hypothesis is done so that its sub-hypotheses are expanded and explored in parallel. In the third method (*Pos&Neg*), the results for the parallel computation of the evaluation parameters (e.g. covered positive examples and covered negative examples) are obtained. The number of branches affects these methods; if there are more branches and more independent hypotheses, more parallelism is achieved. In the *KL1* language, a KL program is compiled into another programming language (i.e. C language). The authors tested their approach (i.e. the three methods) on an email classification problem. The experiment's setting was six processors in a shared memory environment. Based on the result of the experiments, they observed that the more categories that need to be classified, the more DOP (degree of parallelism, or amount of parallelism) is achieved for the induce method. In addition, the background knowledge (particularly the constructed bottom clause) highly affects the DOP of the branch method. High DOP was also achieved with the *Pos&Neg* method.

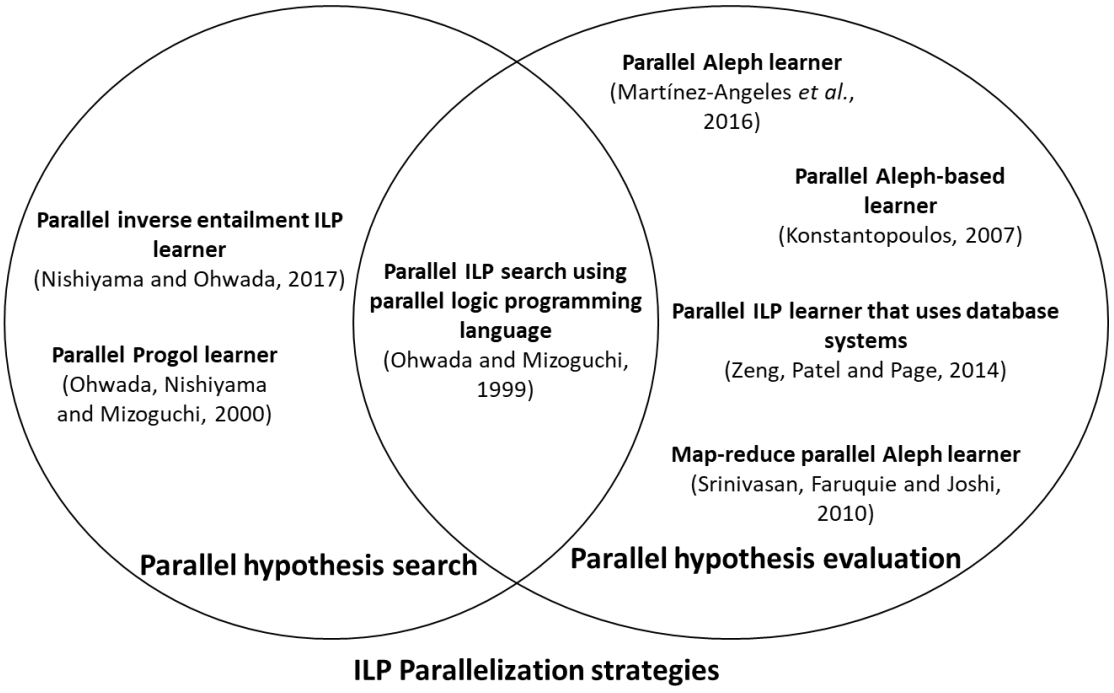


Figure 4.5: A visualization of ILP parallelization strategies.

In the literature, several strategies have been used to accelerate ILP hypothesis search and evaluation. The first strategy is to use a parallel logic programming language, such as the KL1 language, to accelerate hypothesis search and evaluation. The second strategy is to explore the inverse entailment paths in parallel. The third strategy is to improve the hypothesis evaluation by using database engines, such as in QuickFOIL. In the fourth strategy, data parallelism is used to improve hypothesis evaluation – for example, by using MapReduce and GPUs. In the fifth strategy, the learning examples are divided among parallel processors, and each processor learns a weak hypothesis; these weak hypotheses are

then combined into a strong hypothesis. The fifth strategy is a form of ensemble learning. A summary and mapping of ILP parallelization strategies can be seen in Figure 4.5.

4.3 DL parallelization strategies

In this section, we review the parallelization approaches used to accelerate DL-related computations – that is, parallel DL reasoning and parallel instance checking (hypothesis evaluation).

In one study (Wu and Haarslev, 2012), the authors proposed a parallel reasoner using shared memory architecture and a framework for the $\mathcal{ALC}$ DL language. They called their reasoner "Deslog", and it consists of four layers: pre-processing, reasoning engine, post-processing, and the infrastructure layer. The pre-processing layer is responsible for loading the semantic web data (RDF/OWL) into the reasoning engine. The reasoning engine is where the reasoning is performed, and it consists of two components: service provider and rule applier. The service provider includes services for consistency, satisfiability, subsumption, and classifications, whereas the tableau (rule) applier performs the lower level services: conjunctions, disjunctions, universal restrictions, and existential restrictions during the reasoning process. The third layer is the post-processing layer, where the reasoning results received from the reasoning engine are further processed. The infrastructure layer is responsible for providing utility services that support other layers.

The framework works in these steps. First, the pre-processing layer loads and transforms the data. Second, the reasoning engine performs the reasoning (in parallel). Thereafter, it returns the results to the user through the post-processing layer. The reasoner uses different data structures (i.e. lists instead of the traditional trees) to represent concepts and roles to achieve a parallelism-friendly data storage. The list (reasoner main data structure) contains four placeholders: a unique identifier, the DL operator (e.g. conjunction, disjunction, restriction), and two other placeholders that are pointers to other similar data structures depending on the DL operator and its nestedness. A single list represents a single concept, and multiple lists can represents role restrictions (universal or existential) on concepts. To provide better access (through indexing) to roles and instances, a pool of instances and a pool of roles are employed. See Figure 4.6 for an overview of the framework.

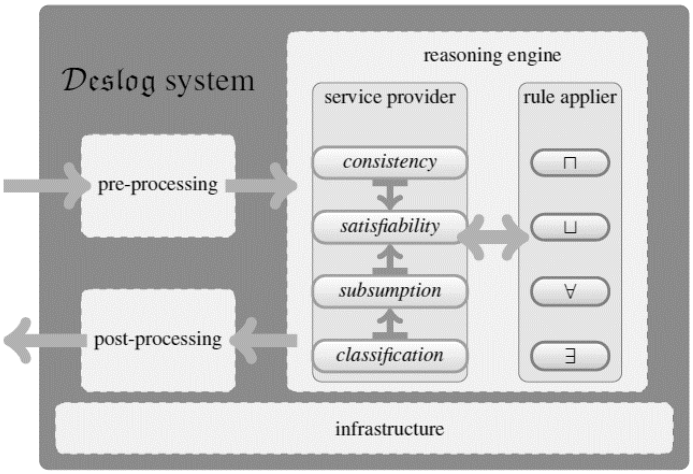


Figure 4.6: An overview of the *Deslog* Framework (Wu and Haarslev, 2012).

In other research (Meissner, 2009), the author proposed a parallel reasoner for the $\mathcal{ALC}$ DL language. The author mapped the reasoning (particularly, inference rules) into a program in Oz language, which is a concurrent (distributed) programming language inspired by languages such as Prolog. The author mapped atomic concepts and roles into *Atoms* and *tuples* for assertions and descriptions of complex concepts and roles. The Mozart system (Oz's development system) is responsible for developing and running Oz programs (Mozart System, 2018). Mozart conducts distributed computing by having a *manager* and a *worker*. The manager assigns work to idle (available) workers and then receives the result back from them. A *worker* processes a specific task assigned by the manager (e.g. answering rules) and reporting the results back. The parallel reasoning algorithm constructs the search (reasoning) tree and then processes the tree nodes (rules) in parallel in a distributed manner through the Mozart system by means of an Oz program. The parallel approach was tested by varying the number of workers linked through network, where each worker was a separate machine. The experiments conducted by the author found that a speedup can be achieved and that the amount of speedup varies depending on the learning problem.

In another study (Chantrapornchai and Choksuchat, 2018), the authors proposed a GPU-accelerated framework for RDF query processing – namely, TripleID-Q. The TripleID-Q framework works in five steps. In the first step (*load*), the RDF file is broken down into four files: one is binary and three are hash tables. One of those files holds the RDF triples, where each of the attributes (i.e. subject, predicate, and object) is a hash key pointing to its respective hash table (file) (i.e. one of the three files). In the second step (*transform*), the query is transformed from string format to numeric format (hash keys). In step three, the binary triple file is loaded into the GPU memory. Fourth, the GPU answers the transformed query, and the results are returned. Fifth, the query results (list of transformed triples) are transformed back into their original format (e.g. string). See Figure 4.7 for an overview of the TripleID-Q framework. The authors also discuss the handling of other queries such as union queries (between queries), join queries, and filter queries.

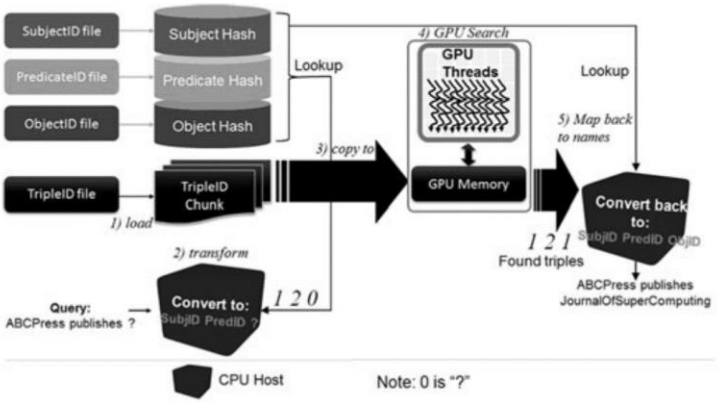


Figure 4.7**: An overview of the TripleID-Q framework (Chantrapornchai and Choksuchat, 2018).**

Several strategies can be used to accelerate DL computations. The first strategy is to use a concurrent logic programming, such as the Oz language, to parallelize reasoning tasks; this strategy is similar to the ILP strategy that uses the KL1 language. For the second strategy, the reasoning paths (or branches) for the tableau algorithm are explored in parallel. In the third strategy, the instance checking (a reasoning task) is parallelized by using data parallelism – GPUs, in particular.

4.4 Analysis

The aforementioned ILP and DL parallelization strategies have demonstrated their effectiveness at speedup computations, although there are limitations. For ILP strategies, the use of a parallel logic programming language reduces the implementation complexity and improves the learning performance; this improvement, however, introduces a communication overhead between the learning algorithm and the parallel logic language. The parallel exploration of inverse entailment paths improve performance, but it is highly dependent on the knowledge base used. The use of database systems to accelerate hypothesis evaluation transfers the evaluation complexity to an existing scalable query answering engine; this transfer however suffers from the overhead of evaluating a single hypothesis, which can introduce a performance bottleneck when there are many hypotheses that need to be evaluated at the same time.

In terms of data parallel strategies, the limitation depends on which framework is used – distributed (e.g. MapReduce) or shared memory (e.g. GPUs). The ILP use of distributed data parallelism is able to handle massive datasets. However, distributed parallelism for ILP systems limits their performance by the cost of network communication, work distribution and load balancing overheads. For ILP data parallelism in a shared memory, the cost of work distribution and load balancing are trivial compared to the distributed parallelism since the parallel process exists within the same machine; here, the performance is limited by the available computing power and the memory space within a single machine. For ILP ensemble learning, the aggregation of weak hypotheses into a single strong one may not always produce good results because of the exact procedure employed for hypotheses aggregation.

For DL strategies, as with the first ILP strategy, the use of a concurrent or parallel logic programming language introduces communication overhead in addition to the complexity of mapping the data into the language specific format. The second strategy relies on parallel exploration of reasoning paths, which are largely affected by the dataset; this DL strategy is similar to the ILP strategy for parallel exploration of inverse entailment paths. Moreover, similar to ILP strategies, data parallelism is also used to speed up DL computations by accelerating instance checking using GPUs.

On an abstract level, both ILP and DL roughly employ the same parallel strategies with slight differences in implementation details. Parallel strategies of both ILP and DL can also be combined to improve the ILP learning and DL-specific computations – especially for DL-based ILP systems. Further performance improvements can also be achieved by using non-parallel strategies, such as query packs in combination with appropriate hypothesis language constraints. However, sampling techniques will not be used in this thesis, since the focus is on the parallelization of ILP computations only. From the literature review, the following observations can be made:

- Most parallel ILPs in the literature use (multi-core) CPUs in either a shared or a distributed (networked) configuration; the use of GPUs is limited.
- Many parallel ILP learners are accelerated versions of the Progol/Aleph classical learners (Nishiyama and Ohwada, 2015; Ohwada, Nishiyama and Mizoguchi, 2000; Konstantopoulos, 2007; Srinivasan, Faruquie and Joshi, 2010; Martínez-Angeles *et al.*, 2016).
- Many parallel ILP learners focus on accelerating a single part of the ILP learning process; that is, either parallel hypothesis evaluation or parallel hypothesis search. Parallel ILP learners that accelerate multiple ILP parts at the same time are rare.
- In DL, parallel DL reasoning tasks are only accelerated using (multi-core) CPUs; in other words, the use of GPU-acceleration to speed up DL reasoning tasks (besides instance checking) is rare.

A summary of ILP and DL parallelization strategies and a mapping of the parallelization strategies to their parallel hardware can be seen in Figure 4.8 and Figure 4.9.

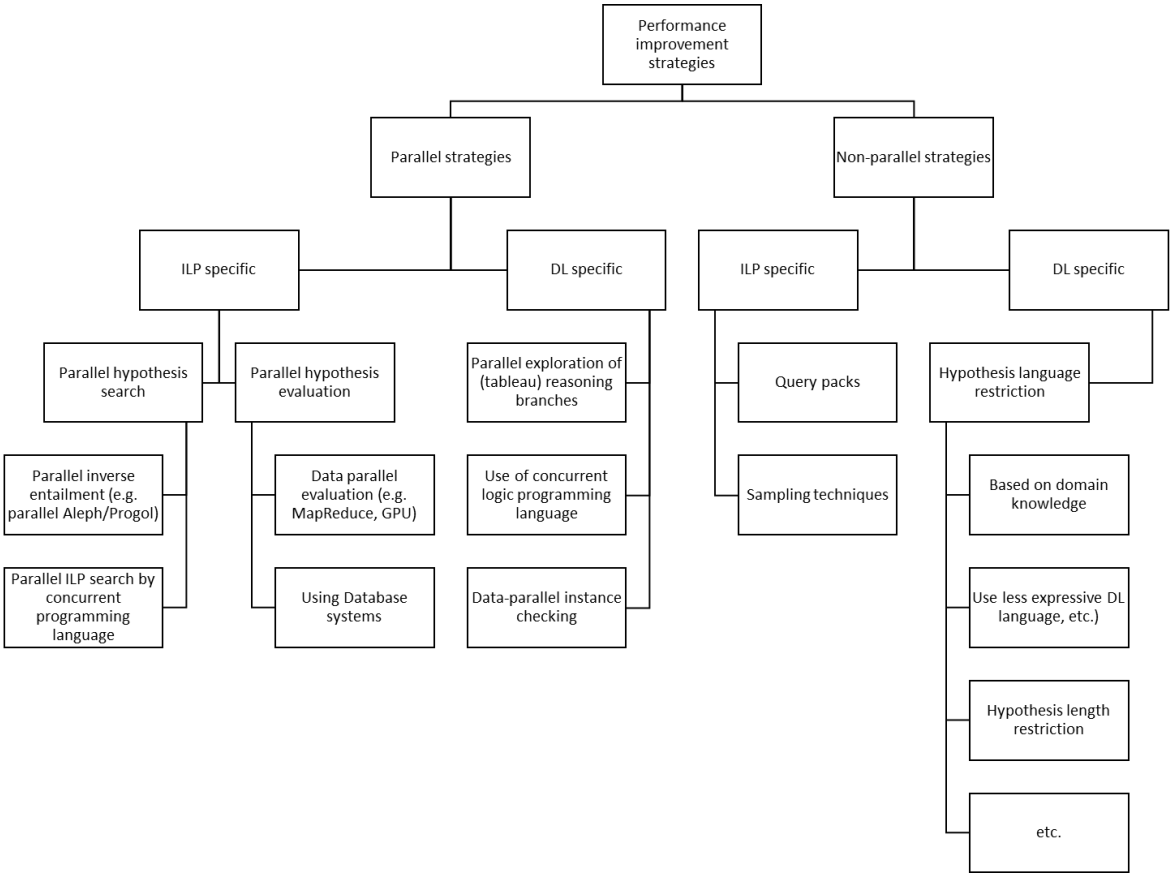


Figure 4.8: A summary of ILP and DL performance improvement strategies.

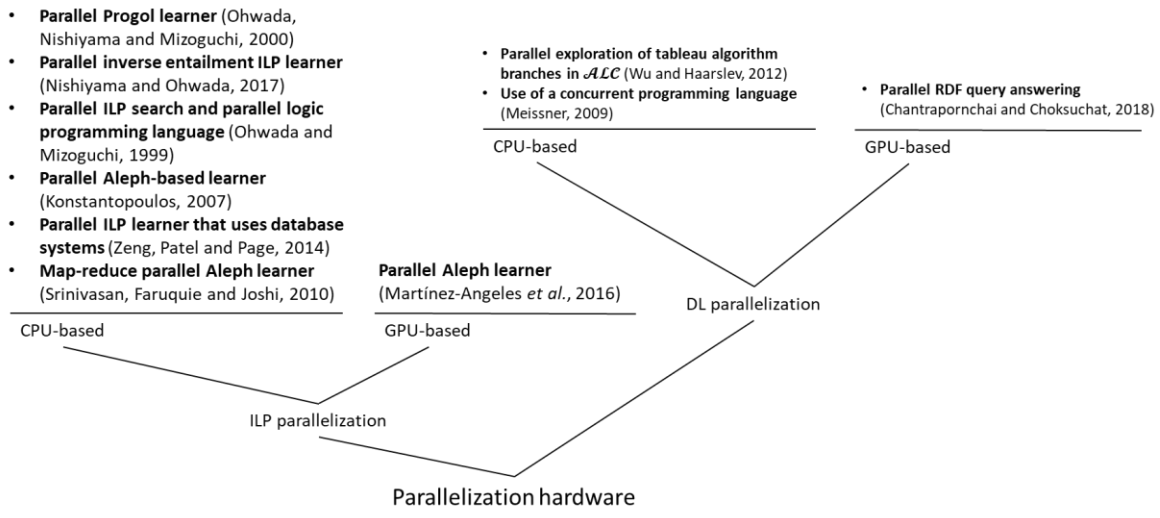


Figure 4.9: A summary of parallelization hardware for ILP and DL.

4.5 Summary

In this chapter, we review the relevant literature for ILP and DL parallelization strategies. For ILP, parallel computing has been successfully used to accelerate either a single part or multiple parts of the ILP learning process (i.e. computations related to evaluation of hypotheses and hypotheses search. For DL, parallel computing has demonstrated its success at accelerating DL reasoning, and parallel

computing could be used to speed up TBox materialization for DL-based ILP learning. Parallel DL reasoning (including GPU-accelerated instance checking) provides a strong opportunity to speed up DL-based (ILP) learning – especially for DL hypothesis evaluation. The following important findings are observed in the literature:

- Data parallelism strategies improve performance in both ILP and DL regardless of their variations (i.e. shared or distributed).
- Parallel exploration of search paths improves performance in
    - Reasoning
    - Hypothesis search
    - Hypothesis evaluation
- Parallel DL-based ILPs are non-existent.

In light of the above findings, the data parallel strategies and parallel exploration of branches are of great importance to this work because of their success in accelerating both ILP and DL computations. Given that a parallel DL-based ILP does not currently exist, in the following chapters we propose a scalable and parallel DL-based ILP learner, SPILDL (**S**calable and **P**arallel **I**nductive **L**earner in **DL**). The proposed learner accelerates the evaluation of hypotheses using multiple GPUs, as discussed in chapter 5. It also utilizes multi-core CPUs for the parallel exploration of the hypotheses space as discussed in chapter 6. Chapter 7 introduces SPILDL-ES as a variation of the main learner targeting high performance – namely, DL-based ILP learning for embedded systems.

Chapter 5: Multi-GPU hypothesis evaluation engine

In the previous chapter, we review and evaluate the parallel and non-parallel strategies to accelerate ILP and DL computations; according to the findings of that review, the combination of several strategies into a single learner is rare – especially combinations that involve non-parallel strategies. In this chapter, we propose a hypothesis evaluation engine for ILP learning in DL. The proposed engine relies on the data parallel strategy – in particular, with GPUs as its backbone strategy. It employs other strategies as well to increase efficiency. In comparison with the approaches and strategies in the literature, the proposed engine has the following combined advantages:

- Direct support for expressive DL languages, such as $\mathcal{ALCQI}^{(\mathcal{D})}$.
- Specially designed to be integrated with the state-of-the-art OCEL algorithm.
- High speed hypothesis evaluation against massive datasets on a single GPU.
- Evaluation of a group of hypotheses in parallel to increase efficiency.
- Scalable because several GPUs can be added to speed up evaluation, locally (same machine) or remotely (network connection).
- Flexible; it can use brand-agnostic mix of GPUs.
- Capability-aware multi-GPU scheduling – for example, by probing the computing power of each connected GPU to determine its scheduling priority.

The result of the above advantages is a high-performance multi-GPU evaluation engine. In relation to the research questions and hypotheses formulated in chapter 1, this chapter addresses the first hypothesis H_1 (Parallel computing reduces hypothesis evaluation time for large DL datasets) and the fourth hypothesis H_4 (Adding more parallel processors reduces ILP learning time). These two hypotheses contribute to answering the research questions: How can scalability issues of DL-based ILP learning be addressed using parallel computing (RQ1)? To what degree can parallel computing improve ILP learning in DL (RQ2)?

We describe our proposed multi-GPU hypotheses evaluation engine, which accelerates DL hypotheses evaluation through a single GPU or multiple GPUs. It can evaluate DL hypotheses expressed in the $\mathcal{ALCQI}^{(\mathcal{D})}$ language – that is OCEL's most expressive refinement operator combined with inverse roles. We first discuss the (DL) knowledge representation details (including the training examples) of the proposed engine. Second, we discuss DL operators (and their algorithmic details) of the target language ($\mathcal{ALCQI}^{(\mathcal{D})}$). Third, we discuss the details of the hypothesis evaluation process for evaluating a single hypothesis. Thereafter, we discuss the details of the multi-GPU evaluation engine. Finally, we provide an evaluation of the proposed multi-GPU engine and the experimental results.

5.1 The proposed multi-GPU engine

An overview of the proposed multi-GPU engine can be seen in Figure 5.1. The details for each component of the proposed engine are discussed in the sections that follow.

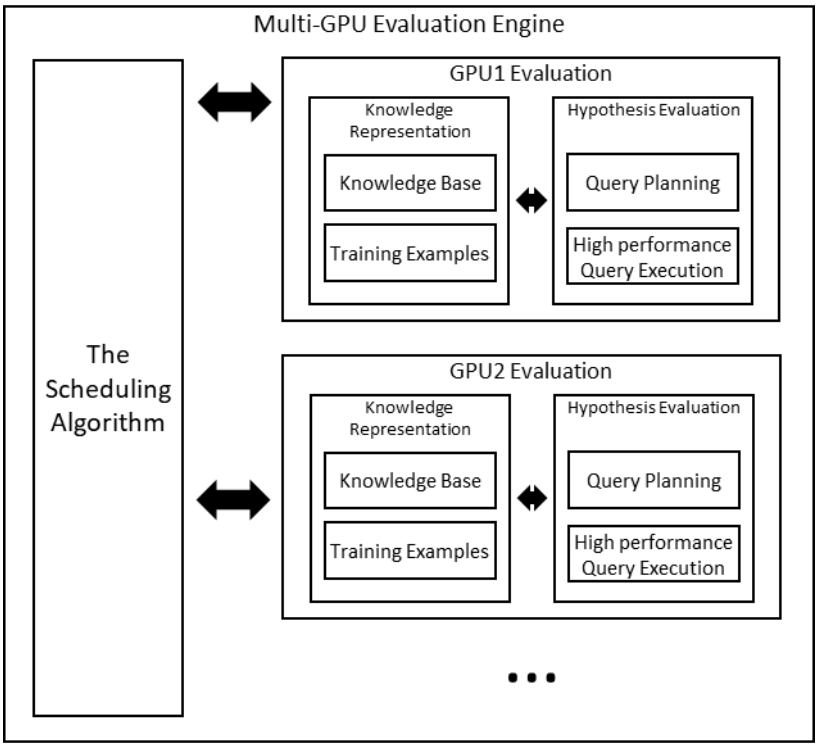


Figure 5.1: An overview of the proposed multi-GPU engine.

It is worth noting that in many learning scenarios, using a single GPU is sufficient to achieve high speed evaluation. However, in some scenarios (e.g. parallel search), a large number of hypotheses is generated; in such cases, a single GPU may not be able to finish evaluating all these hypotheses in a reasonable time. As a result, multiple GPUs can be used to share the hypothesis evaluation load so that the large number of hypotheses is divided among the available GPUs, where each GPU evaluates its assigned hypotheses in parallel with other GPUs.

5.2 Knowledge representation

When using GPUs to accelerate computations, the input and the output data has to be represented natively as a set of matrices (e.g. 1D, 2D, or 3D). Other representations can be used as well; however, the matrix representation is the most efficient for the GPU hardware. As a result, it is the format that was used in this project.

The multi-GPU engine maintains an exact copy of the knowledge base including the learning examples in the dedicated memory for each GPU. The knowledge is represented as an OWL ontology through a set of three 2D matrices: one for the classes and individuals ($size = Individuals \times Classes$), the second for roles or object properties ($size = RoleAssertions \times 3$), and the third for (numeric) concrete roles or data properties ($size = Individuals \times ConcreteRoles$). In order to construct and populate

these matrices, four lookup hash tables are constructed to map the textual (string) names of individuals, classes, roles, and concrete roles to generated numeric values (or IDs). Once these lookup tables are constructed, they are used to map the knowledge base (ABox) assertions into their corresponding 2D matrices. See Figure 5.2 for the three matrices and four lookup tables for the knowledge representation. The details of the three matrices are discussed in turn below.

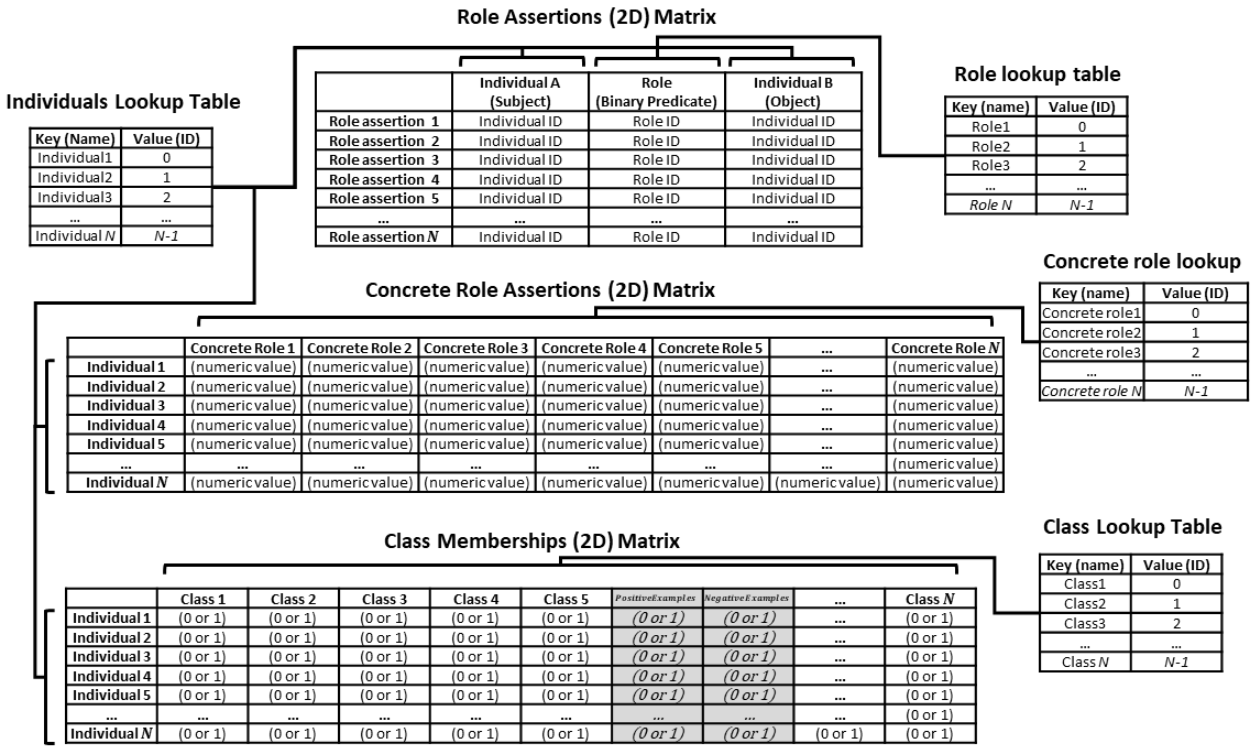


Figure 5.2: All the 2D matrices and their data dependencies.

5.2.1 Class membership matrix

The class membership matrix represents the class assertions (or memberships). In order to construct this matrix, two lookup tables for individuals and classes are used; the rows of the matrix represent the individual IDs, and the columns represent the class IDs. Each cell in the matrix indicates a class assertion for a specific individual with a binary value (represented as a byte): 1 for a class assertion and 0 for no class assertion. The set of training examples is represented as two additional class memberships, where the positive examples (individuals) are members of the $PositiveExamples$ class, and the negative examples (individuals) are members of the $NegativeExamples$ class.

5.2.2 Role assertions matrix

The role assertions matrix represents the role assertions as a set of triples: the subject individual, role, and object individual. The matrix uses the lookup tables of individuals and roles in its construction. The individuals lookup table is used to map the role assertions' subjects and objects into their numerical IDs, and the roles lookup table maps the roles' names to their numerical representation (ID). Each row in the matrix represents a single role assertion. The role assertion matrix is sorted in ascending order

based on the role column to achieve high (GPU) memory performance because of a coalesced memory access pattern.

5.2.3 Concrete role assertions matrix

The class memberships and the role assertions matrices are sufficient to represent knowledge bases expressed in the $\mathcal{ALC}$ language. However, in order to represent knowledge bases in the $\mathcal{ALCQI}^{(\mathcal{D})}$ language, an additional 2D matrix is needed. As a reminder, the $\mathcal{ALC}$ language is a common DL language able to express many real-world concepts. The $\mathcal{ALCQI}^{(\mathcal{D})}$ language is able to express all the hypotheses generated by OCEL refinement operators. The concrete role assertions matrix is designed to represent concrete role assertions. This matrix intends to represent only numerical and Boolean concrete roles and their assertions. In order to construct this matrix, a lookup table is created (and used), which maps concrete role names to their numerical IDs. The layout of this matrix is similar to the class memberships matrix; however, instead of the classes, the concrete roles are in the matrix's columns. In addition, each cell of this matrix indicates a (single) concrete role assertion, which links a specific individual to a specific concrete role; each matrix cell is a signed float number that represents the numeric value of a concrete role assertion for an individual. To indicate no assertion for an individual on a (specific) concrete role, the corresponding cell has the float "*NaN (Not a Number)*" as a value.

The reason for supporting only numerical concrete roles currently is because the OCEL refinement operators used by the proposed learner (described in chapter 6) only support generating refinements for numerical concrete roles.

The main advantage of this (knowledge) representation design is that it represents DL knowledge bases in a GPU-friendly format. In addition, this (knowledge representation) design exhibits higher memory performance (or bandwidth) because of a coalesced memory access pattern. Because of this design, two assumptions related to DL knowledge representation are enforced, the unique name assumption (UNA), and the closed world assumption (CWA); these two assumptions were discussed in chapter 2.

In the next section, we describe the supported DL operators that can be used to construct the candidate hypotheses and that can be evaluated (against the matrices) by the proposed engine.

5.3 Supported DL operations

The proposed (multi-GPU) engine supports DL hypotheses expressed in the $\mathcal{ALCQI}^{(\mathcal{D})}$ language. The following DL operations (including negation and inverse roles) can be used to construct hypotheses:

- Conjunctions
- Disjunctions
- Existential restrictions
- Universal (value) restrictions
- Role cardinality restrictions (MIN, EXACTLY, and MAX)

- Concrete role (data property) restrictions: MIN, EXACTLY, and MAX

The output of applying any of these operations is the set of covered individuals, which is represented as a single binary 1D matrix known as the result matrix/array. The size of this matrix always equals the total number of individuals in the knowledge base. Each index in the result array corresponds (maps) to a specific individual ID; and the value indicates if that particular individual is covered by the given DL operation (value 1) or not (value 0). See Figure 5.3 for an example of evaluating a single conjunction and its respective result array. The array indexes (of matrices) are processed in parallel (see Data parallelism in section 2.3.3).

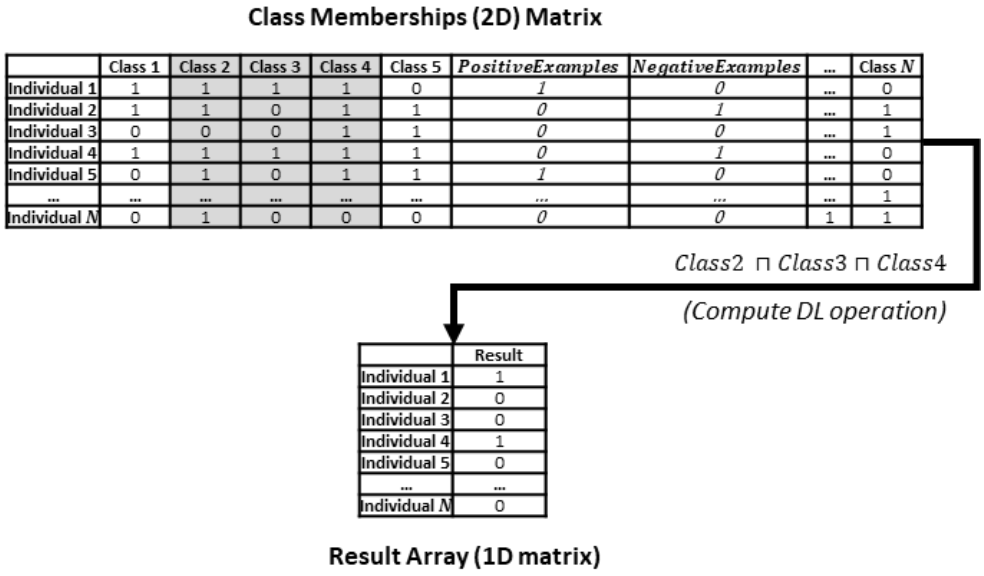


Figure 5.3: An example of evaluating a single conjunction (DL) operation.

In the next subsections, we describe the algorithmic details of each supported DL operation.

5.3.1 Conjunctions and disjunctions

The conjunction and the disjunction operations are similar in their computational steps; however, there are two key differences. First, for conjunction, the result array is set to 1 (true) for all individuals before applying conjunction. For disjunctions, the result array is set to 0 (false) for all individuals before applying disjunction. The reason for the arrangement is that the same result array may be used several times to store intermediate results of other DL operations. As a consequence, the result array has to be reset to all 0s or all 1s (depending on the DL operation) for all individuals to remove previous results before applying any DL operation.

The second difference is the operation type where the "*AND*" operator is used for conjunction and "*OR*" for disjunction. A single conjunction/disjunction, which is comprised of simple (atomic) classes and/or complex classes (i.e. results of other DL operations), is processed in a single step (function). The

computation steps (pseudocodes) for conjunction and disjunction are described in Algorithm 5.1 and Algorithm 5.2, respectively.

For computing conjunction/disjunction, a parallel for-loop is created to process all individuals (in the result array). For each individual, its result is initialized to 1 (for conjunction) or 0 (for disjunction). Second, for every simple class in the conjunction/disjunction, its class membership for the given individual is retrieved from the class membership matrix; then, its class membership (a binary value) is XOR-ed with its negation flag. The negation flag is a binary value for every simple class in a conjunction/disjunction indicating that a specific class is negated (value 1) or not (value 0). Once a given individual's class membership is retrieved and then XOR-ed with its (class) negation flag, it then can be AND-ed (for conjunction) or OR-ed (for disjunction) against its (previous) result. Once all the simple classes are processed, the complex ones (i.e. results of other DL operations) are retrieved for the given individual and then AND-ed (conjunction) or OR-ed (disjunction) against their (previous) result. In terms of the pseudocodes for the two operations, their differences are observed in lines 2, 5, and 10 (on the two pseudocodes). In the pseudocodes, AND, OR and XOR, reflect a logic operation with implementation-agnostic syntax. For example, in C/C++ language, AND is mapped to &&, OR to ||, and XOR to $!=$.

Algorithm 5.1: The pseudocode for the conjunction operation.

Conjunction (with negation)

```
        Input: class membership matrix M, list of simple operands C_op,
        each operand in C_op, is a pair of (Class C ∈ M, negation binary
        flag is_neg), list of complex (intermediate results) classes IR_op,
        each operand in IR_op, is a result array
        Output: result single column (binary) array R

1       Parallel for every individual indv in R
2           R[indv] = 1
3           //compute conjunction on simple classes
4           for every simple operand so(C, is_neg) in C_op
5               R[indv] = R[indv] AND (getMembership(M,indv, so.C) XOR so.is_neg)
6           End for
7
8           //compute conjunction on complex classes
9           for every complex operand co in IR_op
10              R[indv] = R[indv] AND co[indv]
11          End for
12
13      End parallel for
```

Algorithm 5.2: The pseudocode for the disjunction operation.

Disjunction (with negation)

```
        Input: class membership matrix M, list of simple operands C_op,
        each operand in C_op, is a pair of (Class C ∈ M, negation binary
        flag is_neg), list of complex (intermediate results) classes IR_op,
        each operand in IR_op, is a result array
        Output: result single column (binary) array R

1       Parallel for every individual indv in R
2           R[indv] = 0
3           //compute disjunction on simple classes
4           for every simple operand so(C, is_neg) in C_op
5               R[indv] = R[indv] OR (getMembership(M, indv, so.C) XOR so.is_neg)
6           End for
7
8           //compute disjunction on complex classes
9           for every complex operand co in IR_op
10              R[indv] = R[indv] OR co[indv]
11          End for
12
13      End parallel for
```

5.3.2 Existential and universal restrictions

For computing all role restrictions, atomic operations are used to ensure data integrity. In addition, a utility function $setArrayValues(ResultArray, value)$ is used to prepare (initialize) the result array for the restriction computations. This particular function sets a single $value$ for all indexes (individuals) in the $ResultArray$. The pseudocodes of the existential and the universal restriction can be seen from Algorithm 5.3 and Algorithm 5.4, respectively.

For computing existential (∃) restrictions, the utility function ($setArrayValues$) sets all the values in the result array to 0. Thereafter, a parallel for-loop is used to process every role assertion for the given role. In every (parallel) loop iteration, if the subject (i.e. $indvA$) has already been satisfied (its result = 1), then this loop iteration is skipped. However, if no matching assertion has been found (yet), the class $rb.C$ membership (including its negation flag) is computed for the assertion's object ($indvB$). If the result of this computation is 1, then the result for the assertion's subject ($indvA$) is atomically set to 1 to indicate that this (subject) individual satisfies the restriction (and can be ignored by other parallel threads). The reason for the (conditional) pre-checks is to keep the number of atomic writes at minimum (and to reduce redundant computations) because atomic operations serialize the memory access (by the parallel threads), which could potentially introduce performance bottleneck.

Algorithm 5.3: The pseudocode for existential (∃) restriction on a simple class.

∃ Restriction Simple (with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C ∈ M, negation binary flag
        is_neg)
        Output: result single column (binary) array R

1       Call setArrayValues(R,0) //set all individuals' values to 0
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           r_subject = R[ra.indvA]
4           If r_subject equals 0 //no matching role assertion found (yet)?
5               r_object = getMembership(M,ra.indvB,rb.C) XOR rb.is_neg
6               If r_object equals 1 // matching role assertion found?
7                   atomicExch(R[ra.indvA],1) //atomically set value to 1
8               End if
9           End if
10      End parallel for
```

Algorithm 5.4: The pseudocode for universal (∀) restriction on a simple class.

∀ Restriction Simple (with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C ∈ M, negation binary flag
        is_neg)
        Output: result single column (binary) array R

1       Call setArrayValues(R,1) //set all individuals' values to 1
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           r_subject = R[ra.indvA]
4           If r_subject equals 1 //no unmatched role assertion found (yet)?
5               r_object = getMembership(M,ra.indvB,rb.C) XOR rb.is_neg
6               If r_object equals 0 // unmatched role assertion found?
7                   atomicExch(R[ra.indvA],0) //atomically set value to 0
8               End if
9           End if
10      End parallel for
```

In terms of the pseudocodes, the only difference between the computations for the existential and the universal restriction is observed in lines 1, 4, 6, and 7. When both restrictions are used against a complex class (result), the object of the role assertion is checked against its value in the complex class (result) array. See Algorithm 5.5 for the pseudocode of the existential restriction on a complex class (i.e. a result of another DL operation). See also Algorithm 5.6 for the pseudocode of the universal restriction on a complex class. The only difference between computing a restriction on a simple class and a complex class is observed in the pseudocodes at line 5 (for Algorithm 5.5 and Algorithm 5.6). For an example of computing a single existential restriction and universal restriction, see Figure 5.4 and Figure 5.5, respectively.

Algorithm 5.5: The pseudocode for existential (∃) restriction on the result of another DL operation.

∃ Restriction on result

```
         Input: role assertions matrix RA, the restriction body as a
         (intermediate) result array IR

         Output: result single column (binary) array R

1        Call setArrayValues(R,0) //set all individuals' values to 0
2        Parallel for every role assertion ra(indvA,role,indvB) in RA
3            r_subject = R[ra.indvA]
4            If r_subject equals 0 //no matching role assertion found (yet)?
5                r_object = IR[ra.indvB]
6                If r_object equals 1 // matching role assertion found?
7                    atomicExch(R[ra.indvA],1) //atomically set value to 1
8                End if
9            End if
10       End parallel for
```

Algorithm 5.6: The pseudocode for universal (∀) restriction on the result of another DL operation.

∀ Restriction on result

```
         Input: role assertions matrix RA, the restriction body as a
         (intermediate) result array IR

         Output: result single column (binary) array R

1        Call setArrayValues(R,1) //set all individuals' values to 1
2        Parallel for every role assertion ra(indvA,role,indvB) in RA
3            r_subject = R[ra.indvA]
4            If r_subject equals 1 //no unmatched role assertion found (yet)?
5                r_object = IR[ra.indvB]
6                If r_object equals 0 // unmatched role assertion found?
7                    atomicExch(R[ra.indvA],0) //atomically set value to 0
8                End if
9            End if
10       End parallel for
```

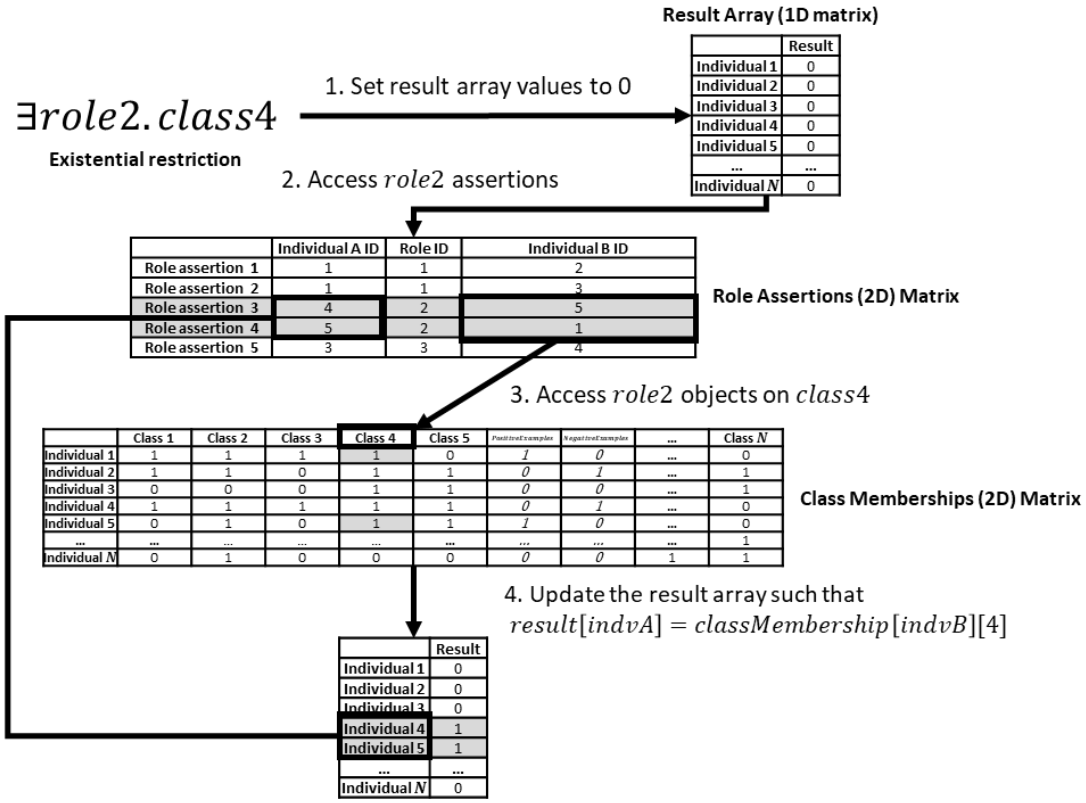


Figure 5.4: An example of evaluating a single existential restriction.

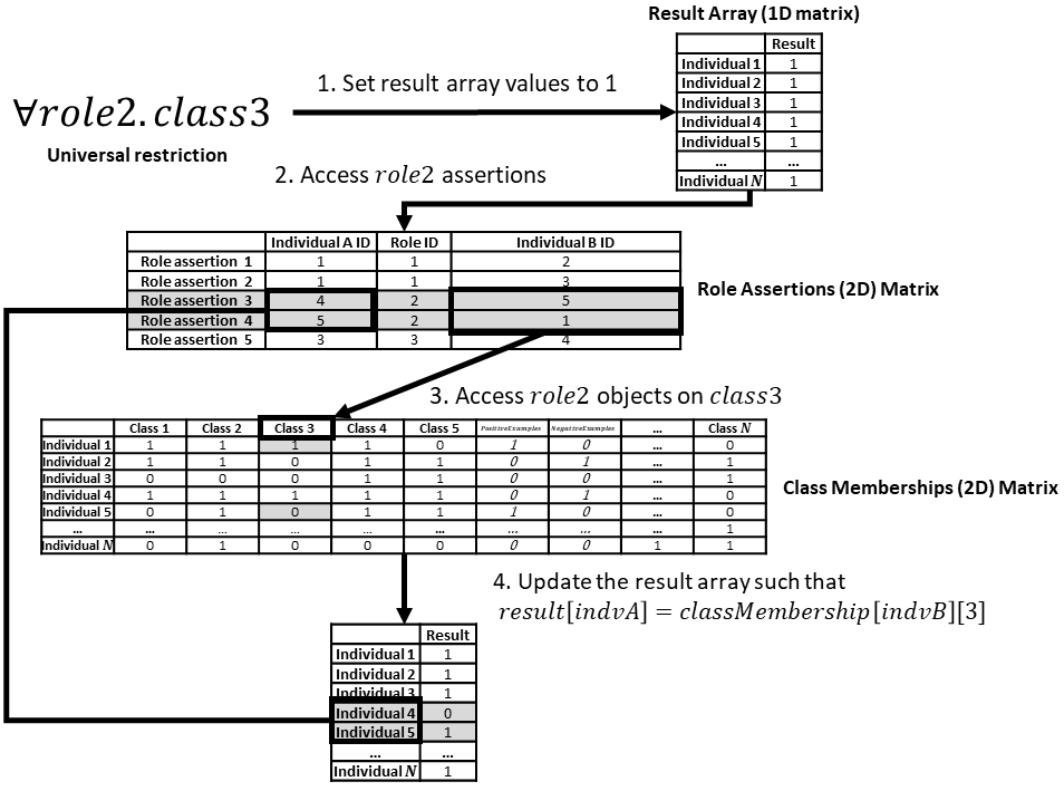


Figure 5.5: An example of evaluating a single universal restriction.

5.3.3 Role cardinality restrictions

In role cardinality restrictions, two conditions have to be met. First, the subject individual must have a matching role assertion. Second, the number of the matched role assertions (for the subject) must conform to the restriction's cardinality (i.e. min, exactly, or max). An example of computing a single role cardinality restriction can be seen in Figure 5.6.

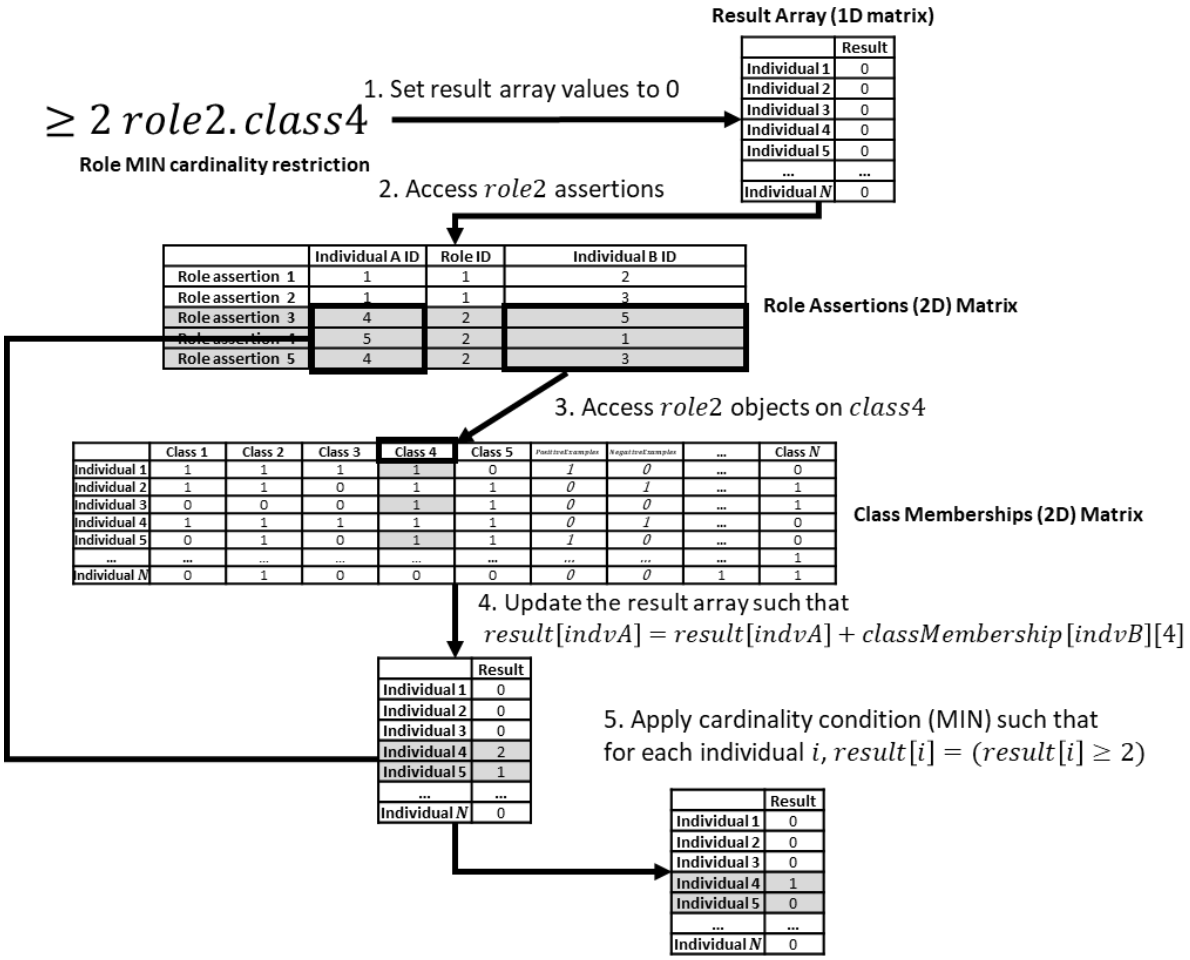


Figure 5.6: An example of evaluating a single role (MIN) cardinality restriction.

The computation of cardinality restrictions is similar to the existential restriction. First, the result array is set to 0. Second, for every matched role assertion for the subject, the subject's value is atomically increased by 1. After all the assertions are processed, another parallel loop is initiated on all individuals in the result array to check if the number of matched assertions for each subject conforms to the restriction's cardinality (min, exactly, or max) and its restriction value. If the cardinality is matched, the value is 1 for the individual; otherwise, it is 0.

For the three cardinality restrictions, the process for the checking and counting matched role assertions is the same. However, the differences between the three cardinalities are observed in their pseudocodes at line 10; see Algorithm 5.7 for their pseudocodes when used on simple classes. When cardinality

restrictions are used on complex classes, only line 3 is changed (for all three cardinalities); see Algorithm 5.8 for cardinality restrictions on complex classes.

Algorithm 5.7: The pseudocodes for the three role cardinality restrictions on simple classes.

Role *MIN* Cardinality Restriction Simple (with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C∈M, negation binary flag
        is_neg), restriction value rv
        Output: result single column (binary) array R

1       Call setArrayValues(R,0)
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           match = getMembership(M,ra.indvB,rb.C) XOR rb.is_neg
4           If match equals 1
5               AtomicADD(R[ra.indvA],1)
6           End if
7       End parallel for
8
9       Parallel for every individual indv in R
10          R[indv] = (R[indv] >= rv)
11      End parallel for
```

Role *EXACTLY* Cardinality Restriction Simple(with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C∈M, negation binary flag
        is_neg), restriction value rv
        Output: result single column (binary) array R

1       Call setArrayValues(R,0)
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           match = getMembership(M,ra.indvB,rb.C) XOR rb.is_neg
4           If match equals 1
5               AtomicADD(R[ra.indvA],1)
6           End if
7       End parallel for
8
9       Parallel for every individual indv in R
10          R[indv] = (R[indv] == rv)
11      End parallel for
```

Role *MAX* Cardinality Restriction Simple (with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C∈M, negation binary flag
        is_neg), restriction value rv
        Output: result single column (binary) array R

1       Call setArrayValues(R,0)
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           match = getMembership(M,ra.indvB,rb.C) XOR rb.is_neg
4           If match equals 1
5               AtomicADD(R[ra.indvA],1)
6           End if
7       End parallel for
8
9       Parallel for every individual indv in R
10          R[indv] = (R[indv]>0 AND R[indv] <= rv)
11      End parallel for
```

Algorithm 5.8: The pseudocodes for the three role cardinality restrictions on complex classes

Role *MIN* Cardinality Restriction on Result

```
        Input: role assertions matrix RA, the restriction body as a
        (intermediate) result array IR, restriction value rv
        Output: result single column (binary) array R

1       Call setArrayValues(R,0)
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           match = IR[ra.indvB]
4           If match equals 1
5               AtomicADD(R[ra.indvA],1)
6           End if
7       End parallel for
8
9       Parallel for every individual indv in R
10          R[indv] = (R[indv] >= rv)
11      End parallel for
```

Role *EXACTLY* Cardinality Restriction on Result

```
        Input: role assertions matrix RA, the restriction body as a
        (intermediate) result array IR, restriction value rv
        Output: result single column (binary) array R

1       Call setArrayValues(R,0)
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           match = IR[ra.indvB]
4           If match equals 1
5               AtomicADD(R[ra.indvA],1)
6           End if
7       End parallel for
8
9       Parallel for every individual indv in R
10          R[indv] = (R[indv] == rv)
11      End parallel for
```

Role *MAX* Cardinality Restriction on Result

```
        Input: role assertions matrix RA, the restriction body as a
        (intermediate) result array IR, restriction value rv
        Output: result single column (binary) array R

1       Call setArrayValues(R,0)
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           match = IR[ra.indvB]
4           If match equals 1
5               AtomicADD(R[ra.indvA],1)
6           End if
7       End parallel for
8
9       Parallel for every individual indv in R
10          R[indv] = (R[indv]>0 AND R[indv] <= rv)
11      End parallel for
```

5.3.4 Inverse role restrictions

The aforementioned role restrictions can be used as inverse role restrictions; by swapping the positions of the subject (*indvA*) and the object (*indvB*) in their (role restrictions) pseudocodes; in other words, the subject is treated as the object, and the object is treated as the subject. A comparison of the existential role restriction and its inverse variation can be seen in Algorithm 5.9.

Algorithm 5.9: The pseudocodes for existential (∃) restriction and its inverse variation on a simple class.

∃ Restriction Simple (with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C∈M, negation binary flag
        is_neg)
        Output: result single column (binary) array R

1       Call setArrayValues(R,0) //set all individuals' values to 0
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           r_subject = R[ra.indvA]
4           If r_subject equals 0 //no matching role assertion found (yet)?
5               r_object = getMembership(M,ra.indvB,rb.C) XOR rb.is_neg
6               If r_object equals 1 // matching role assertion found?
7                   atomicExch(R[ra.indvA],1) //atomically set value to 1
8               End if
9           End if
10      End parallel for
```

Inverse ∃ Restriction Simple (with negation)

```
        Input: class membership matrix M, role assertions matrix RA, the
        restriction body rb, as a pair of (Class C∈M, negation binary flag
        is_neg)
        Output: result single column (binary) array R

1       Call setArrayValues(R,0) //set all individuals' values to 0
2       Parallel for every role assertion ra(indvA,role,indvB) in RA
3           r_subject = R[ra.indvB]
4           If r_subject equals 0 //no matching role assertion found (yet)?
5               r_object = getMembership(M,ra.indvA,rb.C) XOR rb.is_neg
6               If r_object equals 1 // matching role assertion found?
7                   atomicExch(R[ra.indvB],1) //atomically set value to 1
8               End if
9           End if
10      End parallel for
```

In Algorithm 5.9, the differences between the pseudocodes are observed in lines 3, 5, and 7, where subjects and objects are swapped. For line 3, the restriction is applied to the object for the inverse restriction instead of its subject. For line 5, the class membership is checked for the subject in the inverse restriction instead of its object. In line 7, since the inverse restriction is applied to the object, the result array has the restriction result for its objects. An example for computing a single inverse existential restriction can be seen in Figure 5.7.

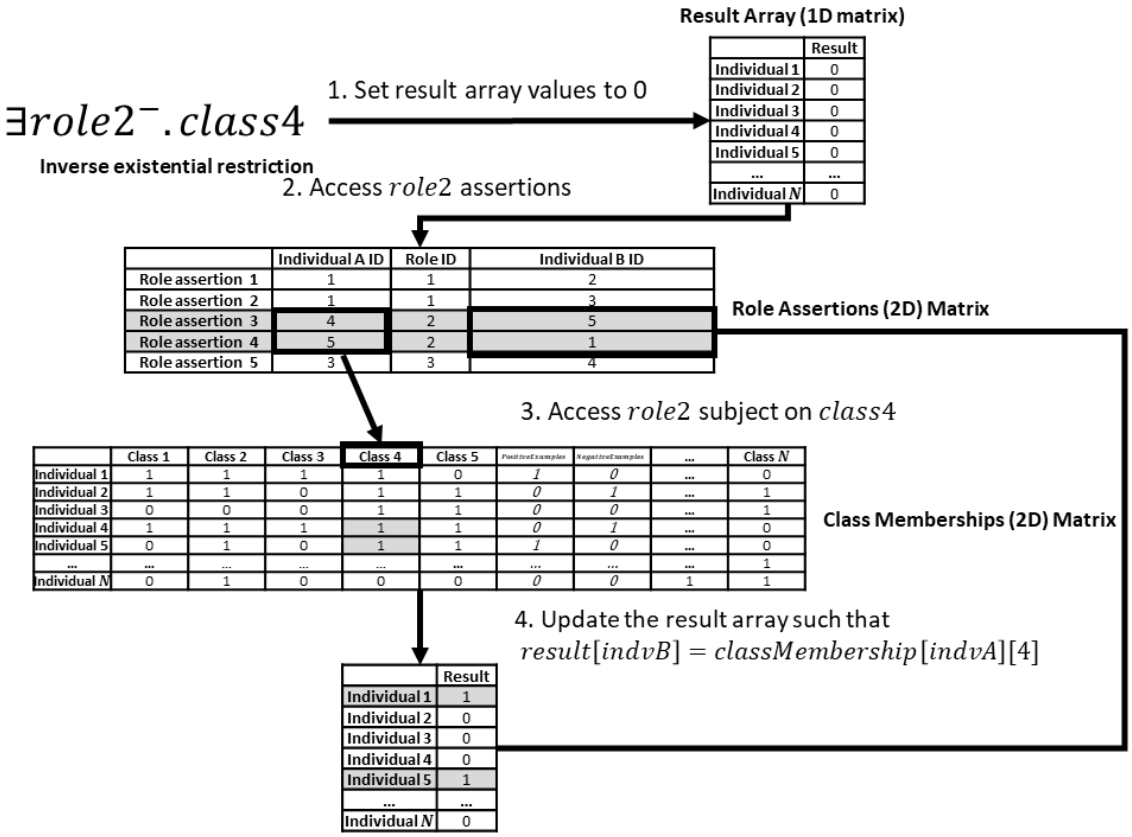


Figure 5.7: An example of a single inverse existential restriction.

Other role restrictions follow the same approach of swapping subjects with objects; for the purposes of brevity, their pseudocodes are not provided here.

5.3.5 Concrete role restrictions

For computing concrete role restrictions, the concrete role value for an individual must conform to the cardinality restriction (and its value). This restriction is computed as follows: First, a parallel for-loop is created to process all the individuals. Second, the concrete role value of each individual is retrieved and then checked to determine whether it is a valid (float) number; if so, the value is checked against the (concrete role) restriction's cardinality and its restriction (float) value. If the cardinality is matched, the value is 1; otherwise, it is 0. The pseudocode for computing the concrete roles' three cardinality restrictions can be seen in Algorithm 5.10. An example for computing a single concrete role restriction can be seen in Figure 5.8.

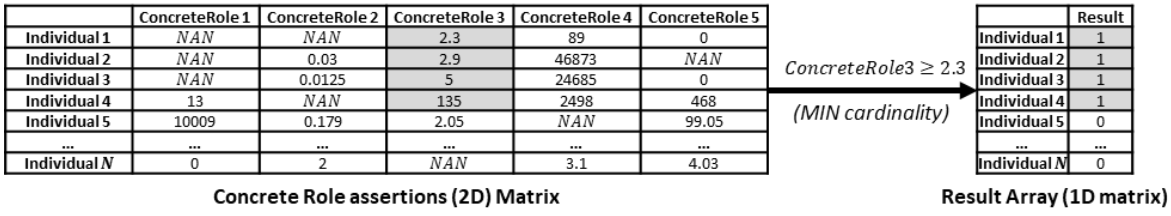


Figure 5.8: An example of a single concrete role restriction.

Algorithm 5.10: The pseudocode for concrete role (data property) restrictions.

Concrete role (data property) MIN Restriction

```
Input: data property matrix D, restriction data property rd,
restriction value (numeric) rv
Output: result single column (binary) array R

1   Parallel for every individual indv in R
2       crVal = getPropertyValue(D,indv,rd)
3       R[indv] = !isNan(crVal) AND (crVal >= rv)
    End parallel for
```

Concrete role (data property) EXACTLY Restriction

```
Input: data property matrix D, restriction data property rd,
restriction value (numeric) rv
Output: result single column (binary) array R

1   Parallel for every individual indv in R
2       crVal = getPropertyValue(D,indv,rd)
3       R[indv] = !isNan(crVal) AND (crVal == rv)
    End parallel for
```

Concrete role (data property) MAX Restriction

```
Input: data property matrix D, restriction data property rd,
restriction value (numeric) rv
Output: result single column (binary) array R

1   Parallel for every individual indv in R
2       crVal = getPropertyValue(D,indv,rd)
3       R[indv] = !isNan(crVal) AND (crVal <= rv)
    End parallel for
```

Each DL operation requires certain parameters and data structures in order to work; see Table 5.1 for an overview of the DL operations and their data structures. Also see Figure 5.9 for a visualization of the DL operations data dependencies.

Table 5.1: An overview of the discussed DL operations and their data structures.

DL Operation	**Needed Data Structures**
Conjunction	• Class memberships matrix • Result array (S)
Disjunction	
Existential (∃) restriction	• Class membership matrix • Role assertions matrix • Result array (S)
Universal (∀) restriction	
Role cardinality restriction	

Concrete role cardinality restriction	• Concrete roles assertions matrix • Result array (S)

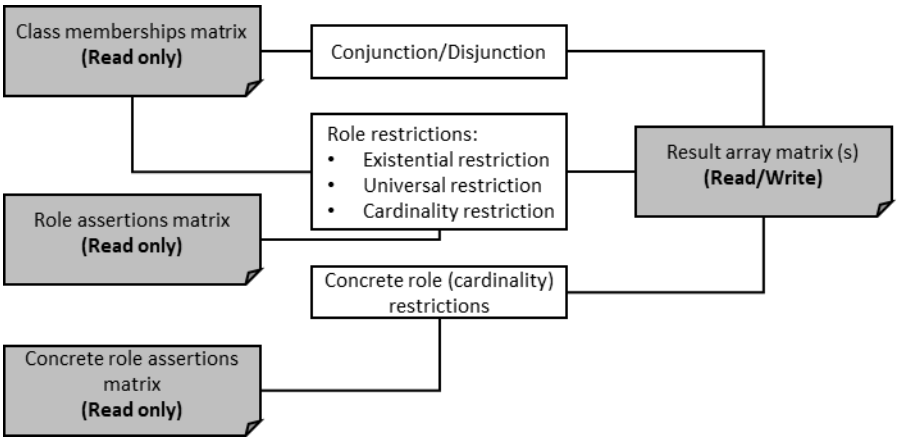


Figure 5.9: An overview of the data dependencies of DL operations.

5.4 DL hypothesis representation and evaluation

In the proposed engine, a complete DL hypothesis (or class expression in OWL) is represented as a tree of DL operations that may contain other nested child DL operations as tree nodes, where the output of a child DL operation is an input for its parent DL operation. This representation is similar to the OWL API's OWL Class Expression.[15] The reason to use such a representation is to promote a fast machine-friendly processing of candidate hypotheses for both evaluation and for the generation of refinements. See Figure 5.10 for an example of a DL hypothesis represented in its (native) tree format and its corresponding textual representation.

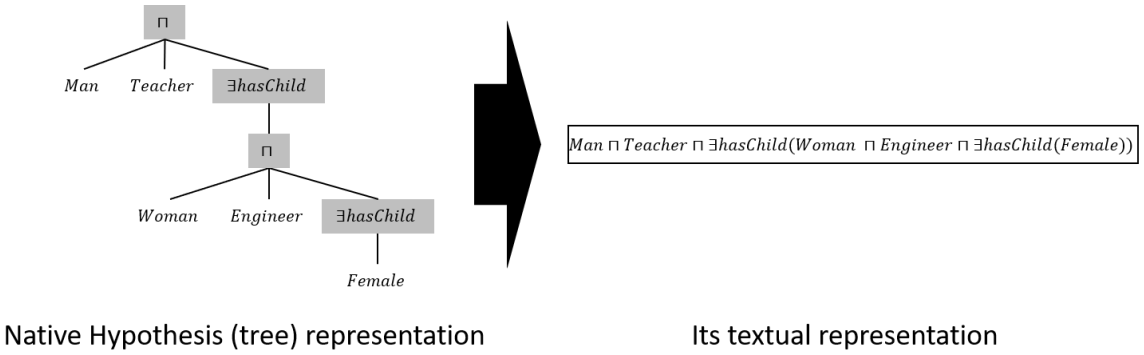


Figure 5.10: An example of the native hypothesis representation and its textual representation.

In order to evaluate a single DL hypothesis, the hypothesis tree is first (recursively) parsed, and a query execution plan (i.e. a sequence of DL operations) is generated. See Figure 5.11 for an example of a DL

[15] https://owlcs.github.io/owlapi/apidocs_4/org/semanticweb/owlapi/model/OWLClassExpression.html

hypothesis and its corresponding query plan. In every generated query plan, an additional operation is added to compute the covered examples (both positive and negative).

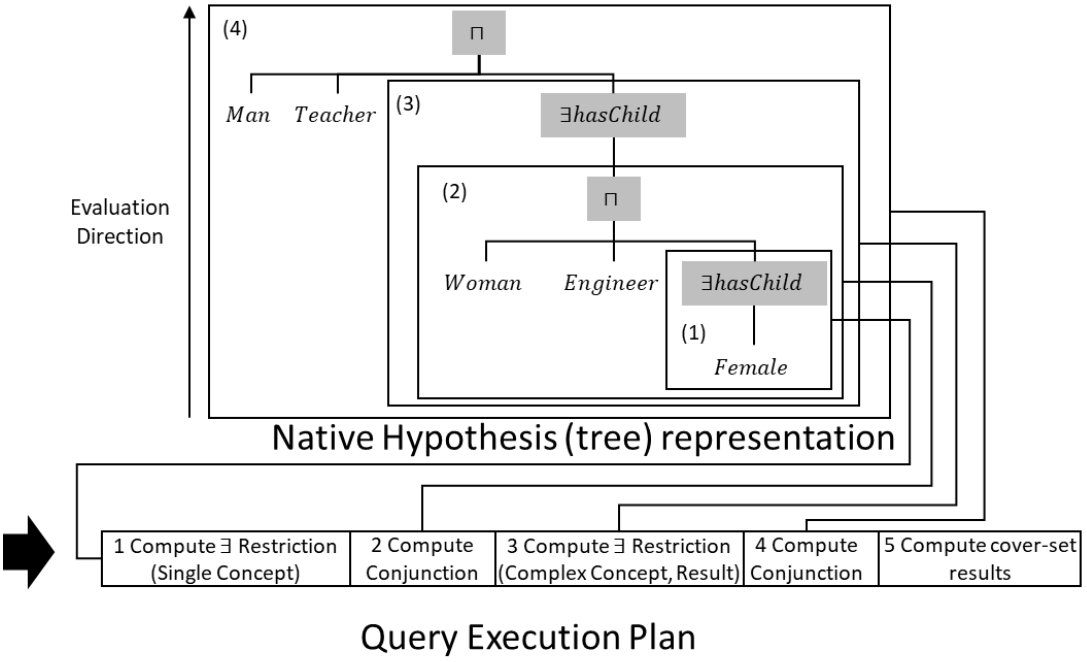


Figure 5.11: An example of generating a query plan for a hypothesis (object).

Once the query plan and its additional operation is generated, it is sent and then executed in the GPU. After the GPU executes its DL operations, it then checks the individuals in the result array against the learning examples. If an individual's value is 1 (i.e. covered by the DL hypothesis) and if it is also a member of the *PositiveExamples* class, it is then considered a covered positive example (and it is counted). The same applies to the negative examples but with the membership of the *NegativeExamples* class. The covered individuals by the hypothesis are checked (and counted) against the positive and negative examples at the same time (in parallel) through a parallel sum implementation. Once all the individuals are processed and once the covered examples are calculated, the coverage result is used to compute the hypothesis score via the OCEL score function (as described in chapter 3). See Figure 5.12 for the evaluation of a single hypothesis.

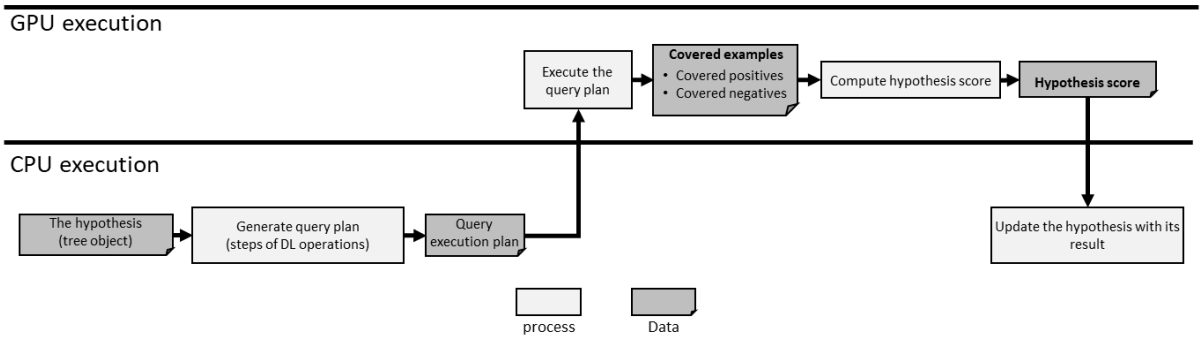


Figure 5.12: The process of evaluating a single DL hypothesis.

In order to reduce the processing overhead for generating query plans and to maximize efficiency, the query engine takes a set of candidate hypotheses as an input and produces a list of evaluation results as an output. First, the query engine generates the query plans for the set of input hypotheses. All the generated query plans are then submitted (all at once) to the GPU's execution queue. Consequently, this batch generation and submission of hypotheses reduces the overhead between creating and then evaluating a single query plan at a time and also maximizes performance gains by keeping the GPU execution queue saturated with computation operations without unnecessary pauses. For an overview of the complete GPU evaluation of candidate hypotheses, see Figure 5.13. Even though these query plans are executed sequentially in the GPU, each DL operation (conjunction, disjunction, etc.) is executed in parallel by the GPU. After a query plan finishes its execution, it updates its corresponding slot (array index) in the covered examples array (with its covered examples). Once the coverage is computed for all the (candidate) hypotheses, their (OCEL) scores are calculated, and then their respective hypotheses' tree objects are updated with their evaluation results.

In an ILP learning scenario, all the hypotheses being processed (at once) typically belong to the same parent (parent hypothesis). In other words, when a hypothesis is refined, all of its generated refinements (child hypotheses) are evaluated by the GPU. See Figure 5.13 for an overview of evaluating candidate hypotheses on a single GPU.

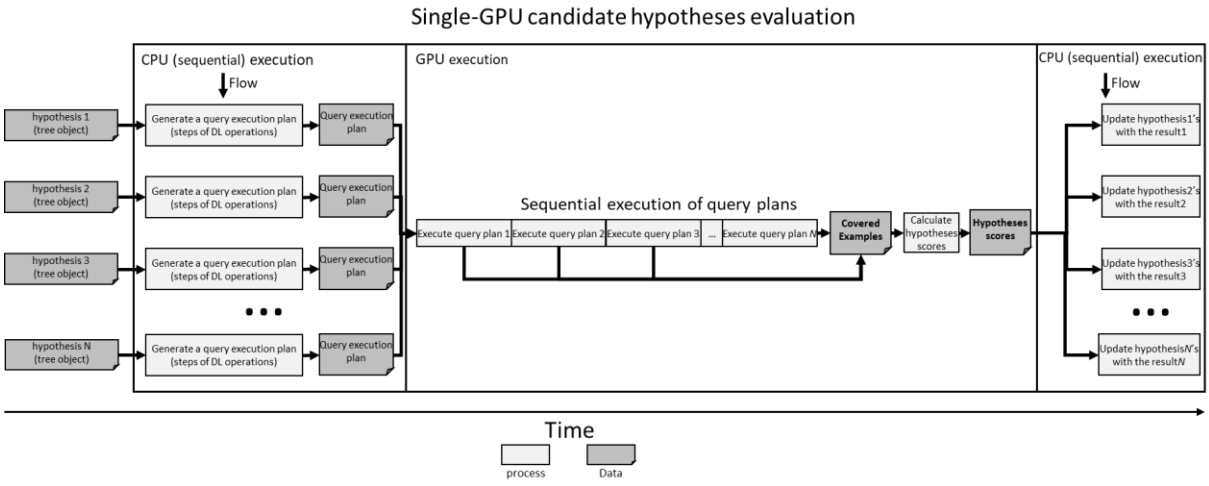


Figure 5.13: Overview of candidate hypotheses evaluation on a single GPU.

5.5 Remote evaluation of hypotheses

In the previous section, we discuss the use of a single local (same machine) GPU for hypotheses evaluation. In this section, we discuss the use of a (remote) GPU that exists in the network for hypotheses evaluation. In remote GPU evaluation, the client (typically an ILP learning algorithm) sends the hypotheses through the network to the server (which has a GPU). The GPU evaluates these hypotheses and then sends their evaluation results back to the client. Initially, the client requests the server's hypotheses evaluation capability. Once the server receives the capability request, it then replies

with its (evaluation) capability; this reply includes information such as the maximum number of hypotheses it can execute at a time and the maximum number of intermediate results arrays, which govern the complexity and the tree depth of a hypothesis object. It also includes additional information, such as the name of the remote GPU (hardware). See Figure 5.14 for the initial (one-time) communication between the client and the server.

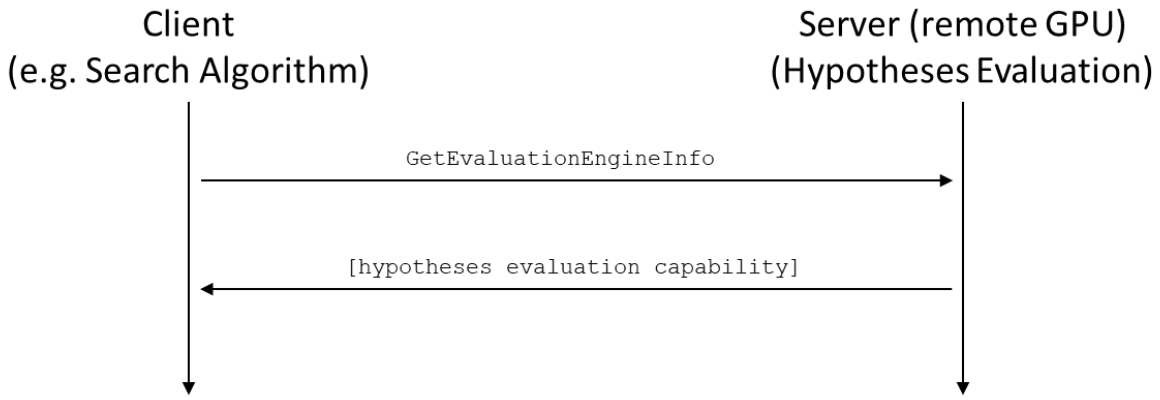


Figure 5.14: The initial communication between the client and the remote GPU (server).

Once the client has information about the server's capability, it can then generate hypotheses for evaluation by this remote GPU. In order for the server to evaluate the hypotheses remotely, the client generates the query plans. Thereafter, it transfers these query plans to the server (one by one). Once the server receives all the query plans, it evaluates them (i.e. computes their coverages and scores). Once the evaluation is done, the server replies back to the client with a list of the hypotheses' evaluation results (their coverages and scores). See Figure 5.15 for a sequence diagram describing the remote evaluation of hypotheses. Before sending the query plans or evaluation results, the sender (client or server) tells the receiving end using "*expect: (data size in bytes)*" the total size in bytes of the data transfer. Once the receiving end (client or server) sees the "expect" message, it replies back with "*ready to receive (same data size in byte)*" to acknowledge (inform) the sender the expected data transfer size; this message is also used by the server or the client when receiving data to verify that all data has been received. It is worth noting that this protocol is an application layer protocol; that is, it can be used on top of another communication link, such as the TCP/IP protocol. However, a consideration is made to the protocol design to ensure that it can use high latency networks, such as satellite networks with reasonable performance by minimizing data packaging. Moreover, since this protocol is an abstract protocol, the details for error detection, correction, and data integrity (checksum) measures are left for the underlying communication link.

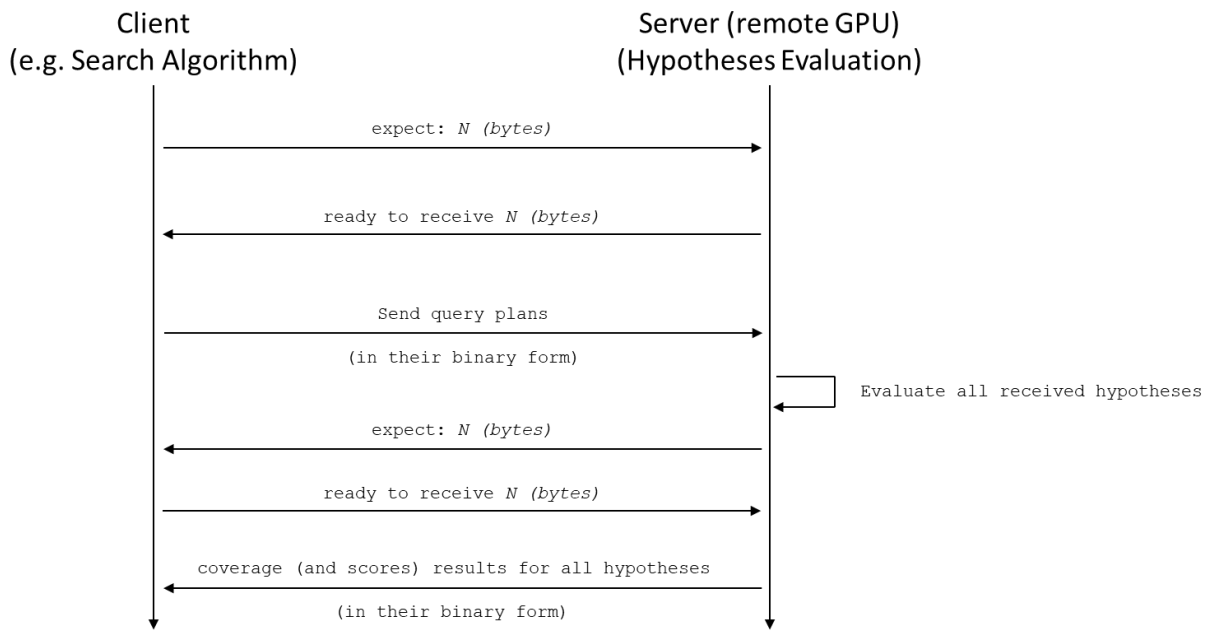


Figure 5.15: A sequence diagram for remote evaluation of candidate hypotheses.

The transfer of query plans and the hypotheses evaluation results through the network is done in binary form to reduce unnecessary overheads of serialization/deserialization, which add to the network communication overhead. Both the client and the server have the same DL dataset. However, the client may only have the TBox if the memory is not sufficient to hold the entire ABox or simply in order to dedicate more memory to store the search nodes. The server, however, holds the entire dataset in memory (i.e. both TBox and ABox).

Because of the nature of this communication protocol, the communication performance is high (even on slower networks) because the transfer of query plans and their results is done in their raw binary form. In addition, this high-level protocol can used on top of other (lower-level) protocols, such as TCP/IP, Bluetooth, ZigBee (or IEEE 802.15.4),[16] and Z-Wave (ITU-T G.9959)[17]. In this work, we limit the implementation to TCP/IP networks for reasons of practicality and brevity.

In the next section, we discuss the use of multiple GPUs for hypothesis evaluation.

5.6 Multi-GPU parallel hypotheses evaluation

In some (ILP learning) situations, a single GPU may not be able to cope with the full load of evaluating hypotheses – especially when there is a high flow of incoming hypotheses that need evaluation. In other words, a single GPU may be overwhelmed and may be unable to evaluate a large number of hypotheses (all at once) in an acceptable timeframe, thus introducing a performance bottleneck for the ILP learning process. To address this problem, we propose a multi-GPU hypotheses evaluation, whereby multiple

[16] https://standards.ieee.org/standard/802_15_4-2015.html
[17] https://ec.europa.eu/eip/ageing/standards/home/domotics-and-home-automation/itu-t-g9959_en

GPUs share the workload of hypotheses evaluation. Each GPU has an exact copy of the knowledge base in its own memory. In order to manage the evaluation of a large number of hypotheses and their distribution among the GPUs, a scheduling algorithm is used. The scheduling algorithm employs static scheduling to determine the distribution of all the hypotheses among the GPUs before evaluating any of them. Thereafter, the scheduler directs the hypotheses to their corresponding GPUs for evaluation. See Figure 5.16 for a comparison of a single GPU evaluation with multi-GPU evaluation (i.e. the massively parallel evaluation engine compared to a single evaluation engine system). There are three variations of the proposed engine: 1) for shared memory architecture, 2) for distributed architecture, and 3) a hybrid of the shared memory and distributed architecture. The three variations are discussed in the next subsections.

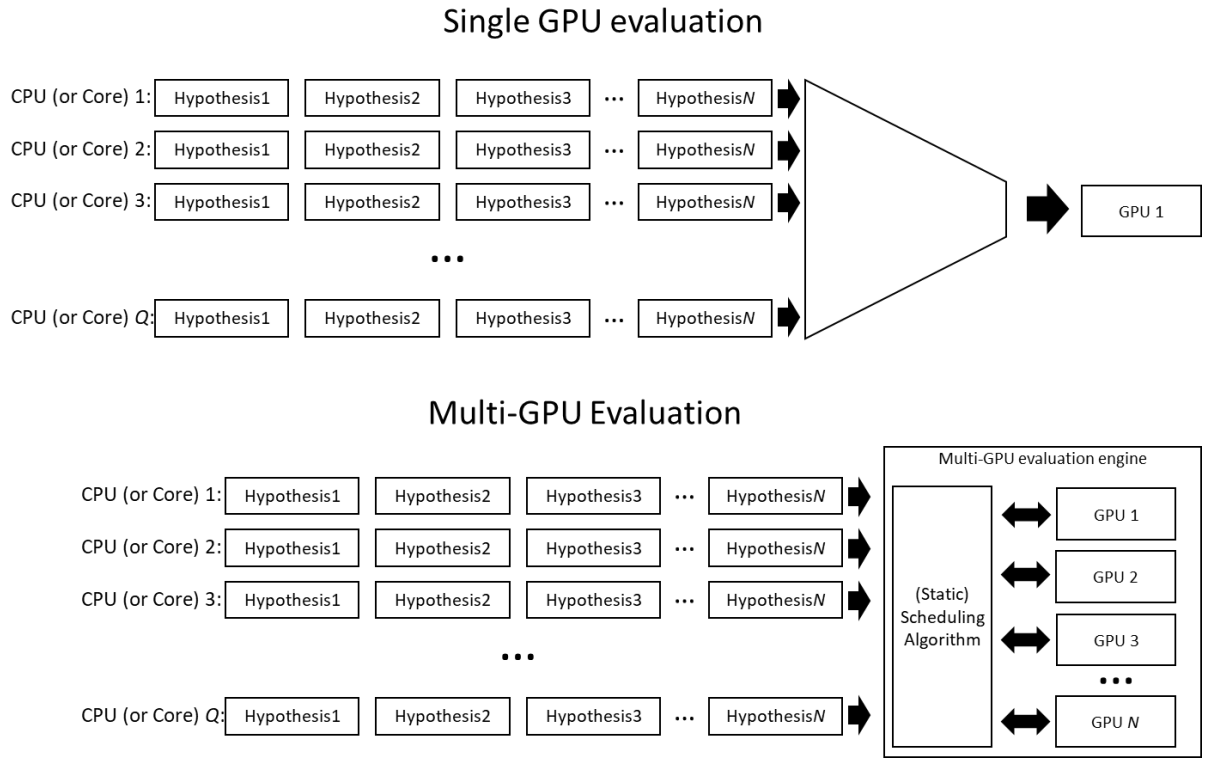


Figure 5.16: A comparison of single GPU and multi-GPU evaluation.

5.6.1 GPU capability-aware scheduling

The scheduler takes into account the computing power of each (available) GPU (including the remote ones). In order for the scheduler to obtain an estimation of a particular GPU's computing power, it issues a probing query for every GPU. The scheduler then measures the probing query's execution time for each GPU; the GPU with the lowest execution time is the most powerful. The probing query is a hypothesis made from conjunction of four concepts in the loaded knowledge base in the GPU's memory. The measurement of the GPU performance is done only once at the start-up of the multi-GPU engine. Remote GPUs' execution times (of the probing query) are slightly longer (than if they were on the same machine) because of their network communication overhead for transferring the query plans

and retrieving the results). See Figure 5.17 for the static scheduling algorithm and its GPU performance measurement. In static scheduling, the work distribution among the parallel processors is determined beforehand, and it remains fixed (static) until all the processors finish their assigned tasks.

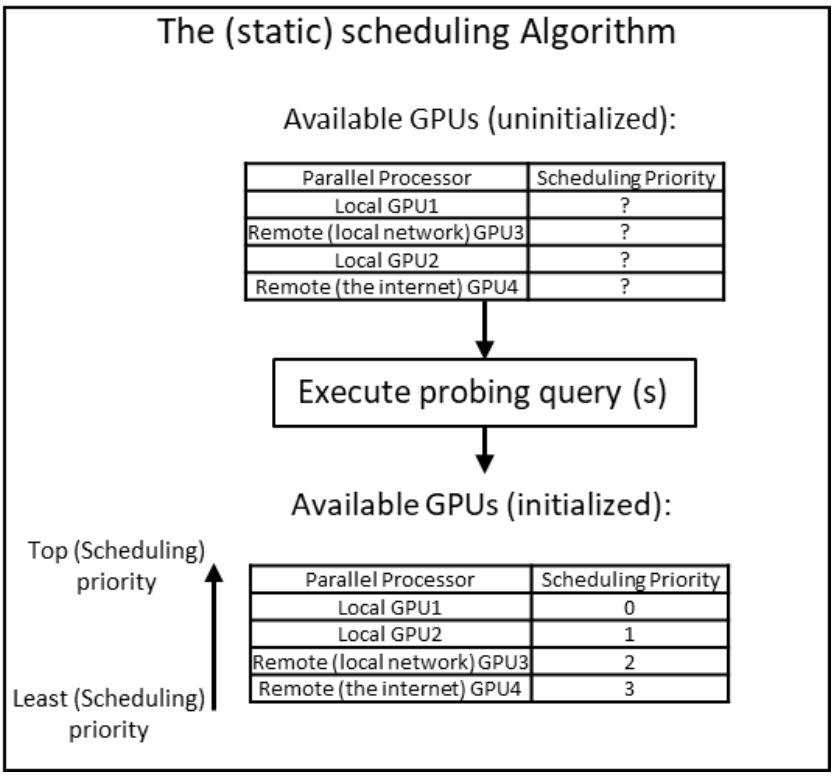


Figure 5.17: The static multi-GPU scheduling algorithm.

The multi-GPU scheduler supports three work distribution policies, which can be selected (and configured) by the user; these policies are gradient scheduling, equal load scheduling and relative load scheduling. These scheduling policies may already exist in the parallel computing literature with different names. However, while most work distribution policies follow similar ideas from an abstract view, the details may slightly change to achieve better results for a particular problem. In the context of this work, the novelty here is to use work distribution polices from parallel computing in addition to the probing query to distribute work among multiple GPUs more appropriately in order to more efficiently speed up ILP-related computations in DL.

Gradient scheduling

In this scheduling strategy, the most powerful GPU does all the computations (up to certain user-defined limit, evaluating at most 10,000 hypotheses at a time). Once this GPU is overwhelmed (i.e. the user-defined limit is exceeded), the additional hypotheses are directed to the next most powerful GPU, which also has its own user-defined upper limit. Once this second GPU is overwhelmed, the flow is directed to the third best GPU and so on.

Equal load scheduling

This scheduling strategy divides the number of hypotheses under evaluation as evenly as possible among available GPUs. For example, if 10,000 hypotheses needed to be evaluated with two available GPUs, the scheduler sends 5,000 hypotheses to each.

Relative load scheduling

This scheduling strategy is similar to equal load scheduling, but the workload division is based on a user-defined ratio. For example, the user may specify the ration (60%, 40%) of load distribution for two GPUs. In that distribution, when 10,000 hypotheses are under evaluation, GPU1 handles 6,000 hypotheses (60%), and GPU2 handles 4,000 (40%) hypotheses. Both the gradient and the relative scheduling policy have user-defined parameters, unlike the equal load scheduling, which simply equally divides all the incoming hypotheses among the available GPUs.

5.6.2 Massively parallel multi-GPU evaluation

The proposed multi-GPU engine can use a heterogeneous mix of local and remote GPUs in order to achieve massively parallel multi-GPU evaluation. In terms of combinations of GPUs, there are three variations: multiple local GPUs (shared memory), multiple remote GPUs (distributed memory), and a heterogeneous mix of local and remote GPUs (hybrid memory). See Figure 5.18 for the three variations.

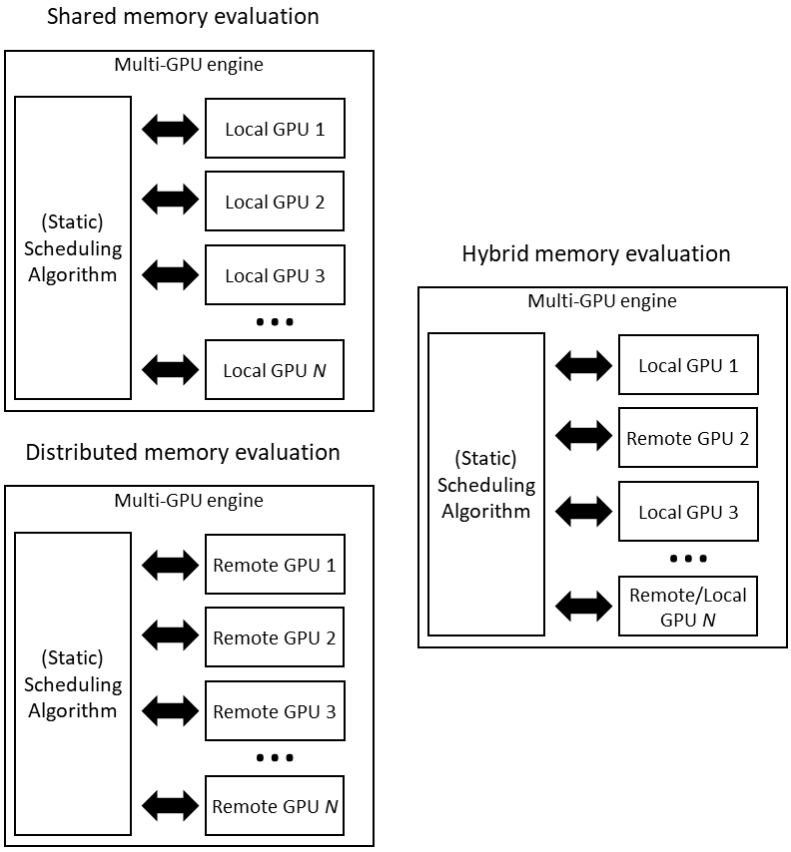


Figure 5.18: The three scenarios for multi-GPU evaluation.

5.7 Implementation

The proposed evaluation engine is implemented using the C/C++ programming language. The rapidXML library[18] is used to parse the XML files of OWL ontologies (represented in OWL/XML). The OpenCL API is used for writing and executing the GPGPU code (in the C language) for hypotheses evaluation. In terms of supported DL operators, inverse role restrictions are not implemented because the OCEL refinement operator does not use them to generate refinements. In order to keep the implementation details concise, the following subsections only discuss major details in knowledge representation, GPGPU implementation, and remote hypothesis evaluation.

5.7.1 Knowledge representation

All knowledge representation matrices (i.e. class memberships, role assertions, and concrete role assertions) are implemented as three (separate) arrays in the row-major representation; that is, an entire matrix row is represented as a single continuous chunk of memory. With this representation in mind, an entire (matrix) row can be retrieved in a (potentially) single memory transaction, which improves the GPU memory access (speed) and therefore improves efficiency. See Figure 5.19 for the class membership matrix and its row-major representation.

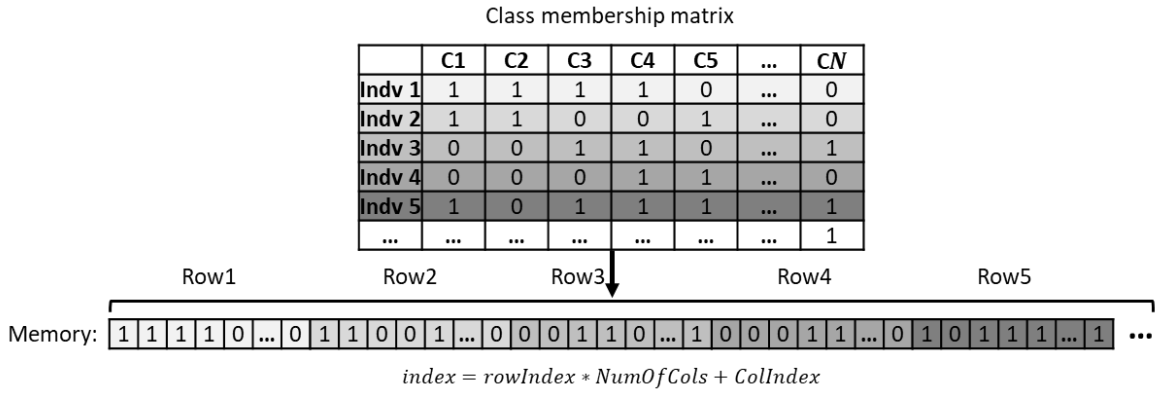
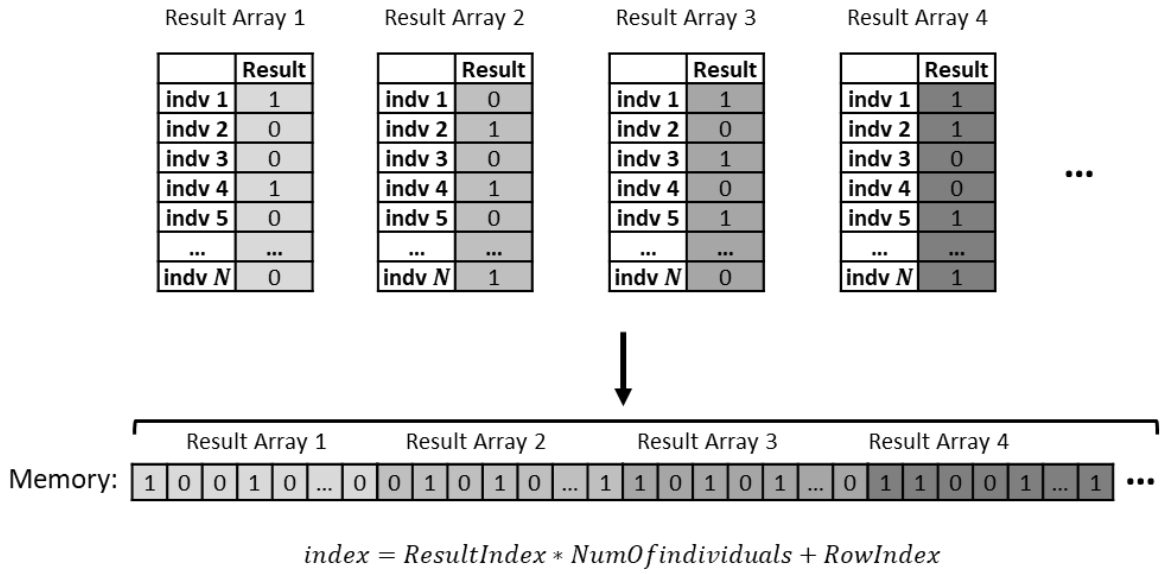


Figure 5.19: The class memberships matrix in the row-major representation.

In Figure 5.19, the entire class membership matrix is mapped into the GPU memory as a single 1D array. The size of the array is the same as the matrix size (i.e. $individuals \times classes$); the mapping formula ($index = rowIndex * NumOfCols + ColIndex$) is used to map the 2D matrix's values into its 1D array representation. In order to represent all result arrays (i.e. result matrices), a single (1D) column-major array is used. The reason for the column-major representation is that only access to a single result array is needed at a time by the GPU to store (or read) the results of a DL operation. See Figure 5.20 for the column-major representation of the result arrays.

[18] http://rapidxml.sourceforge.net/



$$index = ResultIndex * NumOfindividuals + RowIndex$$

Figure 5.20: A result array mapped into the column-major representation.

The choice of a row-major or column-major representation on a particular matrix is based on the memory access patterns of DL operations when they are applied to that (particular) matrix. Regardless of the matrix mapping (row-major or column-major), the mapping formula is computed whenever a memory access is needed on a given matrix. The use of the mathematical operation $a \times b + c$ for the mapping formula is common in many data parallel applications (especially for GPUs in computer graphics). Consequently, many GPUs have a hardware implementation to compute the result of this formula in a single processor instruction (e.g. MAD instruction, or Multiply-And-Add); this instruction (or equivalent) is included in the instruction set of many GPUs (and modern CPUs). With this (processor) instruction in mind, memory access to the mapped matrices has virtually zero overhead.

5.7.2 GPGPU implementation

The reason for choosing the OpenCL library is its flexibility in terms of GPUs brands; OpenCL is an open standard (brand-agnostic). With that in mind, many GPUs (and even multi-CPUs) from different vendors can be used (at the same time) by OpenCL for parallel computing, thus enabling better use of the available parallel hardware (regardless of its vendor). In this work, we limit the application of OpenCL to (multiple) GPUs even though it can run the same GPGPU code in multi-core CPUs.

5.7.3 Remote GPGPU evaluation

The communication protocol used with remote GPUs (discussed in Remote evaluation of hypotheses), is implemented on top of TCP/IP networks. In the initial communication between the client and the server (i.e. the client requesting the server's evaluation capabilities), the client sends its request as a textual (string) message, and then the server replies back with a (small) XML message containing its capabilities. During (ILP) learning, the client generates the query plans for its hypotheses and then

arranges their raw bytes into a series of binary TCP/IP messages. Thereafter, the client sends these messages to the server. See Figure 5.21 for the client transfer of hypotheses to the server.

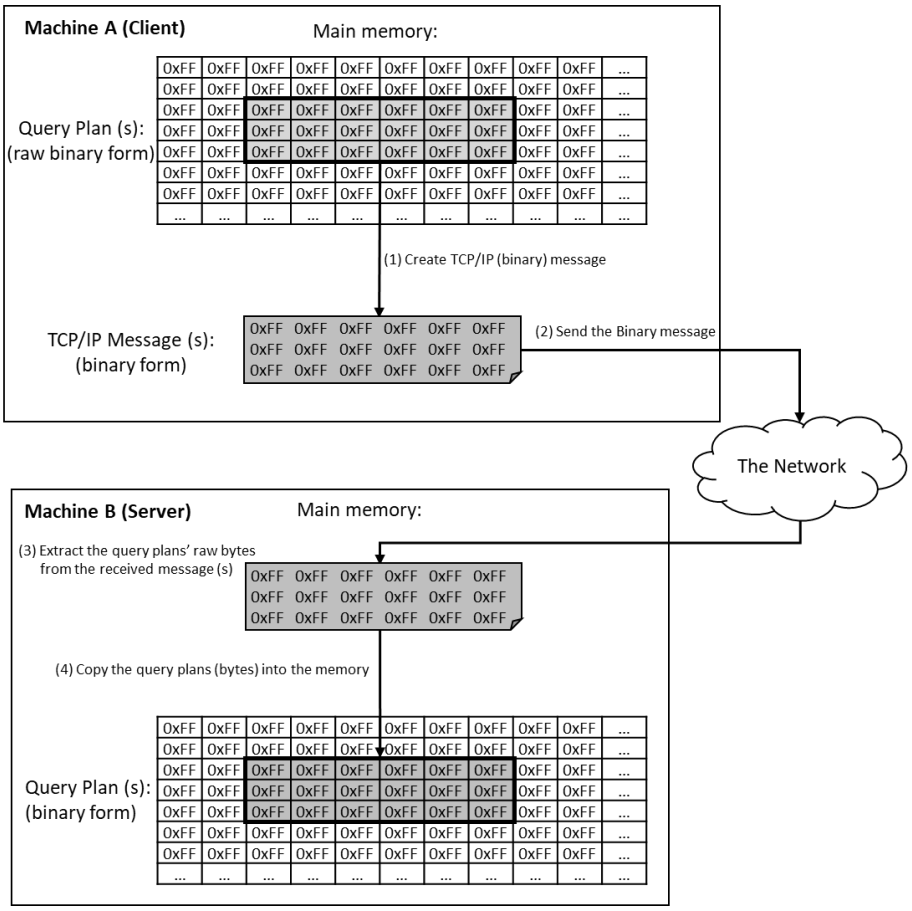


Figure 5.21: The process of sending hypotheses to the server (for remote evaluation).

Once the server receives the (TCP/IP) messages, it extracts the bytes of the query plans. Thereafter, it copies the bytes into its memory, followed by the evaluation of these hypotheses using its GPU. Once the query plans are evaluated, the server arranges their results (bytes) into binary TCP/IP messages and then sends them to the client. The client extracts the results from the received messages and then updates its (remotely evaluated) hypotheses with its (received) results. See Figure 5.22 for the server's communication to the client with the evaluation results.

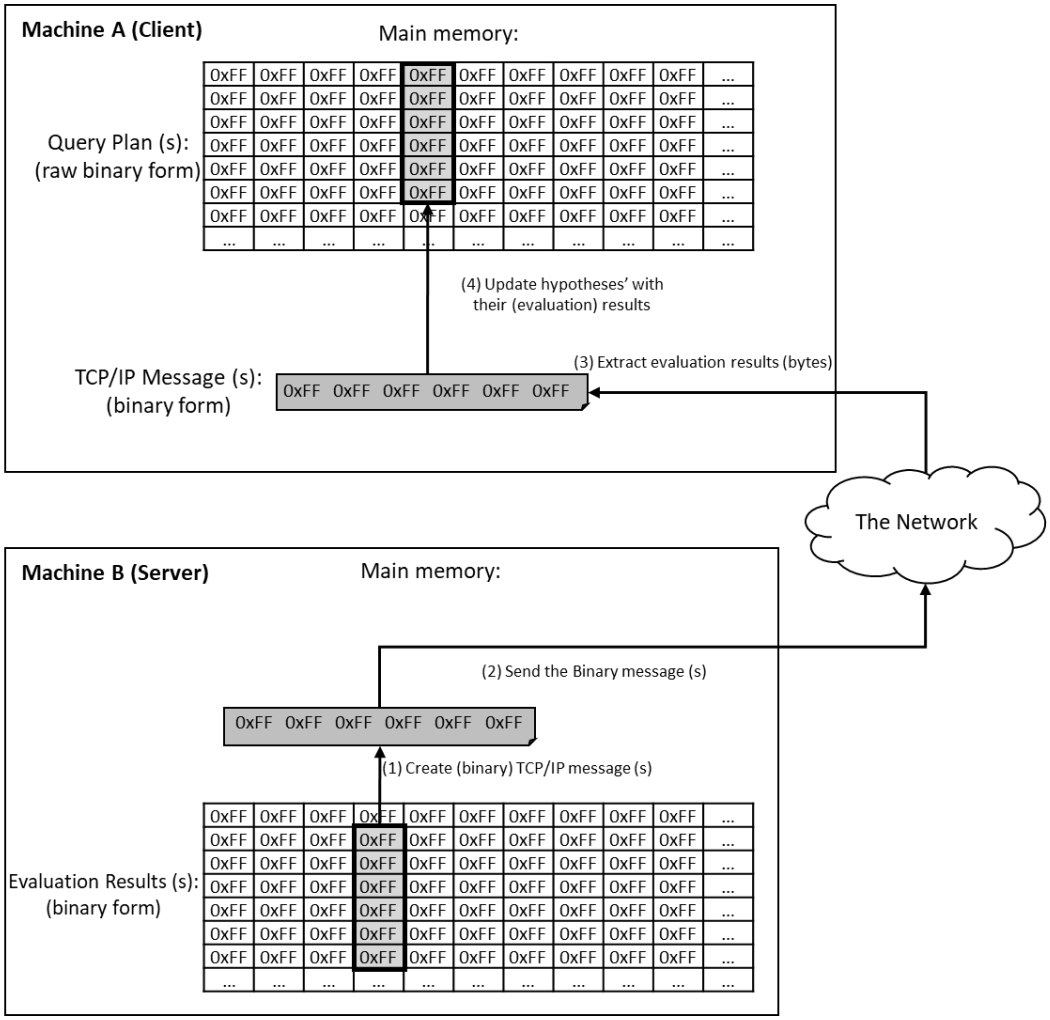


Figure 5.22: The process of sending the evaluation results back to the client.

5.8 Evaluation

The proposed multi-GPU engine is first evaluated on the basis of its individual components and then with regard to the whole engine. The purpose of the evaluation is to address the research hypotheses mentioned at the beginning of the chapter, H_1 and H_4. The evaluation was carried out as follows:

1. Evaluate the individual DL operators.
2. Evaluate single local GPU versus a single remote GPU.
3. Evaluate the complete multi-GPU engine (with its three scheduling policies).

5.8.1 Configuration of the experiments

In this section, we discuss the configuration of the experiments, including the hardware and software used. See Table 5.2 for the specifications of the machine used for the experiments. The CPU clock was reduced to 1.58 GHz (from 3.60 GHz) so that performance gains would not be influenced by the use of high clock CPU.

Table 5.2: The machine specifications for the experiments.

Hardware	**CPU**	Intel Core i9-9900K CPU (locked at default clock of 3.6 GHz on all cores)
	GPUs	• Intel UHD Graphics 630 (within the same chip of the i9 CPU) • Nvidia GeForce RTX 2080
	Main memory	32 GB
Software	**Operating system**	Microsoft Windows 10 (64 Bit)

5.8.2 Evaluation of DL operators

In this section, we report the evaluation of each DL operator separately (i.e. in isolation) in a single GPU (i.e. "Nvidia GeForce RTX 2080") and in a single CPU thread. The single CPU thread was used to establish the performance baseline. The execution times were measured in μs (microseconds), and the average of 10 collected (time measurements) readings was calculated.

Conjunction and disjunction

Conjunction and disjunction are similar, besides using similar logical operations. The performance of both operations is affected by 1) the number of operands (or classes) and 2) the number of individuals involved. In order to study the two factors for both operations, two experiments for each operation were needed. In the first experiment, the operation was evaluated on a fixed number of classes but with varying numbers of individuals. In the second experiment, the number of individuals was fixed, but the number of classes was varied. For the results of the first experiment, see Table 5.3 and Figure 5.23 (for its graph).

Table 5.3: Conjunction and disjunction of four classes with a varying number of individuals.

#Individuals	**Conjunction**		**Disjunction**	
	Single CPU thread	**Single GPU**	**Single CPU thread**	**Single GPU**
10	0.00	60.10	0.00	59.10
100	0.00	53.20	0.00	53.30
1,000	3.00	58.70	4.00	58.90
10,000	37.20	65.20	44.50	65.40

100,000	365.50	205.20	446.90	208.80
1,000,000	3756.50	1640.20	4494.50	1583.40
10,000,000	37574.20	15702.40	45009.60	15178.70

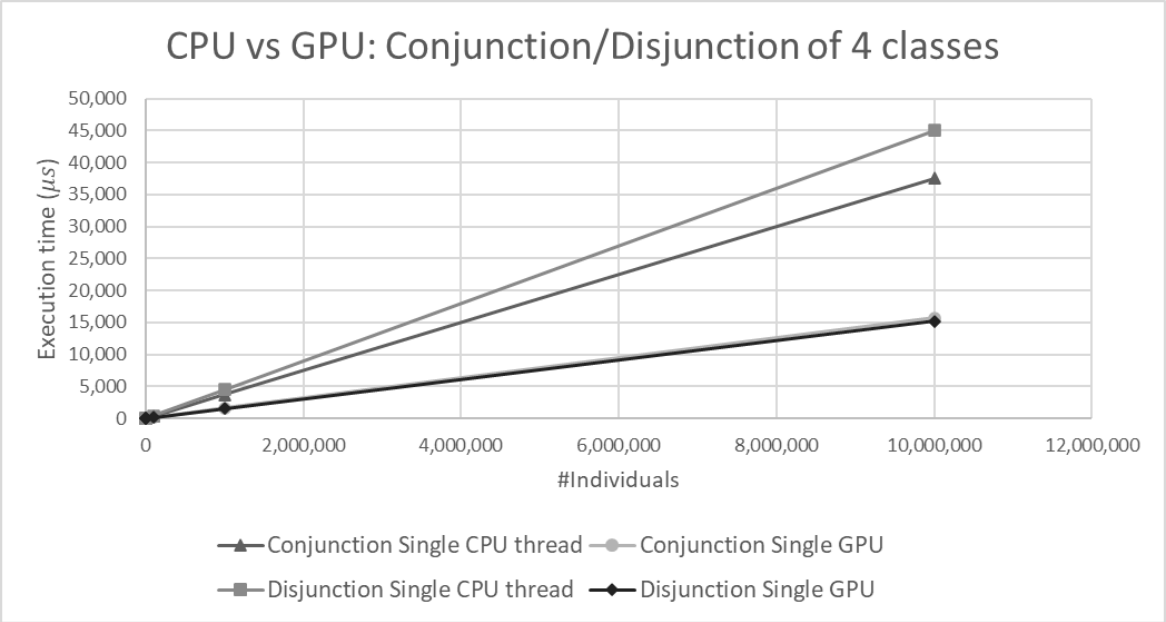


Figure 5.23: Plot of four classes of conjunction and disjunction with a varying number of individuals.

In the second experiment, an upper limit of 31 classes (operands) was enforced. This limit is a general-purpose limit that exhibits a good balance between memory transfer performance (especially for caching) and the complexity of a conjunction/disjunction that can be processed by the GPU. See Table 5.4 for the experiment's results and Figure 5.24 for its graph.

Table 5.4: Conjunction and disjunction on 1,000,000 individuals with a varying number of classes.

#Classes	Conjunction		Disjunction	
	Single CPU thread	**Single GPU**	**Single CPU thread**	**Single GPU**
2	2509.10	1532.60	3353.50	1480.90
4	3955.40	1644.50	4487.20	1593.50
8	6973.70	1910.90	8292.50	1741.40
16	13671.80	2130.90	15504.50	1920.70
31	26322.90	2308.80	28116.00	2123.40

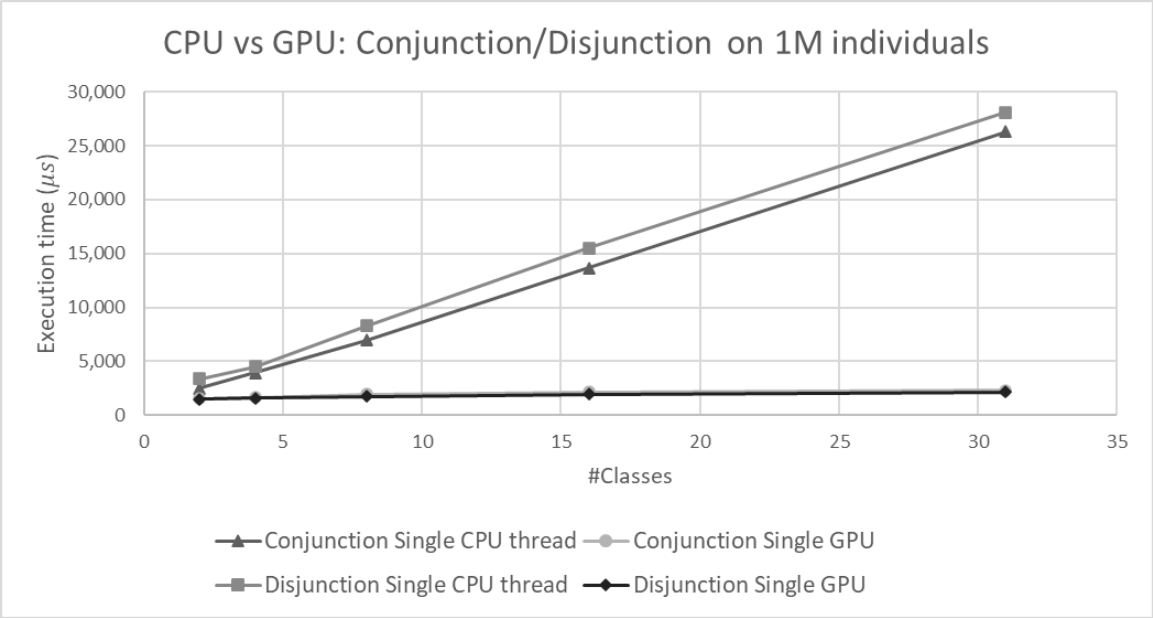


Figure 5.24: Plot of conjunction and disjunction on 1 million individuals with a varying number of classes.

The two experiments yielded the following results:

- The single CPU was faster than the GPU when the number of individuals was $\leq 10{,}000$.
- According to the performed experiments, the GPU always outperformed the single CPU regardless of the number of operands.
- The performance of conjunction and disjunction operation was roughly identical within a GPU. For a single CPU, there is a small performance gap between the two operations (conjunction and disjunction) that grows relative to the number of individuals.

Existential and universal restrictions

Because of the use of atomic operations in role restrictions to avoid data integrity issues (from data race), the performance of role restrictions is highly dependent on the nature of the role assertions data – especially the subject of the role assertion because it affects how many times atomic operations are called. An atomic operation funnels down the parallel memory operations (read/write) into a series of sequential memory operations to ensure data integrity; this transformation of parallel operations into sequential operations, negatively affects the parallel performance in relation to the number of called atomic operations. As a result, two experimental cases per restriction are used to evaluate the two scenarios of role assertion data. In other words, the performance of a restriction always falls between these two evaluation cases. In the first case, role assertions are generated using 20 million individuals, where each individual is the subject of at most one role assertion. In the second case, role assertions are generated for the same subject (but with a changing object). The two cases provide sufficient evaluation of the performance of role restrictions; that is, in real-world applications, the performance of role restrictions theoretically always falls between the boundaries of the two cases. For experiment results,

see Table 5.5 and Figure 5.25 for the existential restriction, and see Table 5.6 and Figure 5.26 for the universal restriction.

Table 5.5: Existential role restriction with a varying number of role assertions.

#Role Assertions	Single CPU thread (Case 1)	Single GPU (Case 1)	Single CPU thread (Case 2)	Single GPU (Case 2)
10	2937.30	273.90	2863.90	284.90
100	2941.10	267.50	2941.60	279.50
1,000	2940.80	284.90	2941.90	272.70
10,000	2972.60	281.80	2993.80	270.50
100,000	3279.30	281.60	3160.90	348.90
1,000,000	10362.60	483.80	5143.60	979.10
10,000,000	91062.90	5116.90	28757.40	7183.50

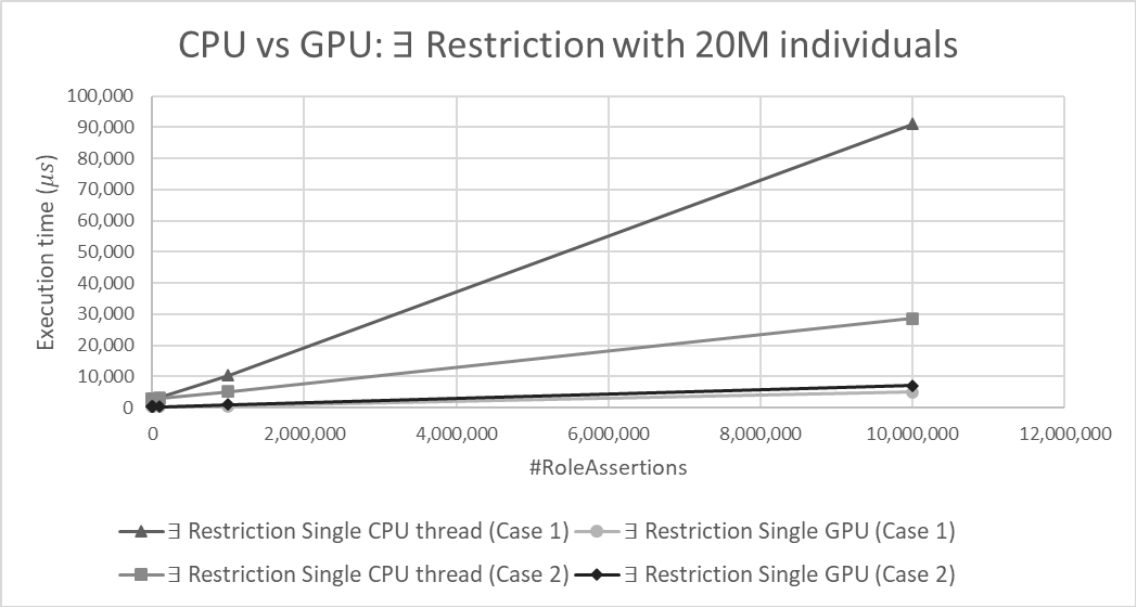


Figure 5.25: Plot of existential role restriction with varying role assertions.

Table 5.6: Universal role restriction with a varying number of role assertions.

#Role Assertions	Single CPU thread (Case 1)	Single GPU (Case 1)	Single CPU thread (Case 2)	Single GPU (Case 2)
10	2882.30	281.70	2950.80	282.70
100	2863.80	271.80	3027.80	282.90
1,000	2945.10	288.40	2943.90	279.00
10,000	2963.20	277.80	2992.10	277.10
100,000	3147.80	287.90	3124.20	287.30
1,000,000	5566.50	339.10	4830.50	330.10
10,000,000	50833.10	1760.80	24456.00	795.90

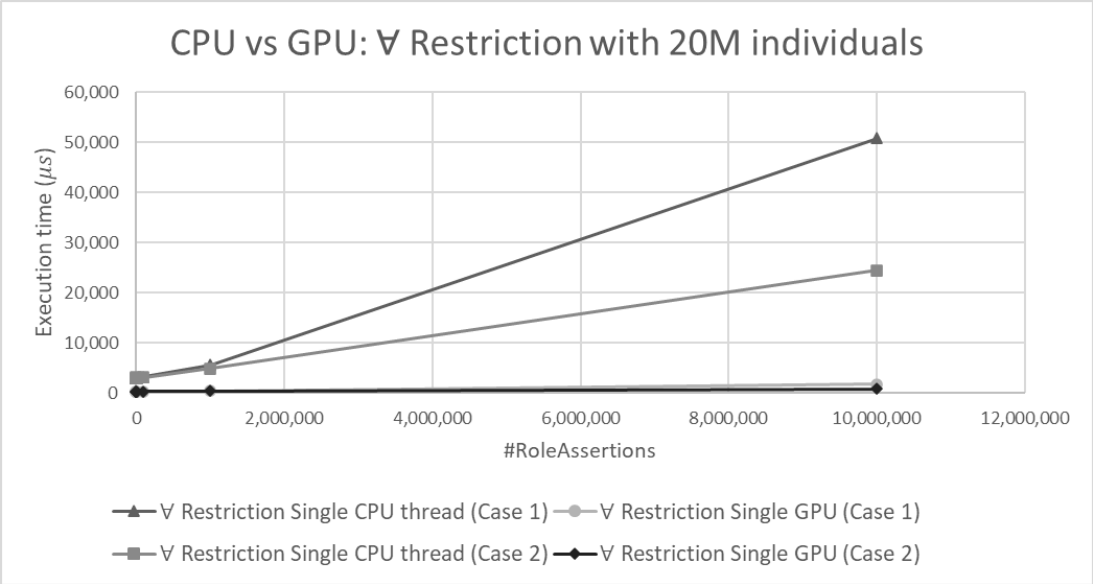


Figure 5.26: Plot of universal role restriction with varying role assertions.

After testing the two scenarios on the existential and the universal restriction, the following observations were made:

- The single CPU has a consistent performance on both cases. However, there is a performance gap between the two restrictions (on both cases) that becomes apparent on 1 million assertions.
- The GPU has outperformed a single CPU on both restrictions regardless of number of assertions, the performance gap (between single CPU and GPU) grows relative to the number of assertions, especially on 10 million assertions.

- On 10 million assertions, The GPU was 18 times faster than single CPU for existential restriction (on case 1, its best-case scenario), and 31 times faster for universal restriction (on case 2, its best-case scenario).

Cardinality role restrictions

Since this restriction is (algorithmically) similar to the previous two role restrictions – especially to the existential restriction – the two experiments discussed earlier are used for every cardinality. See Table 5.7 and Table 5.8 for the experiments for the role cardinality restrictions, and Figure 5.27 and Figure 5.28 for their graphs.

Table 5.7: Role cardinality restriction with a varying number of role assertions on case 1.

#Role Assertions	Single CPU thread (case 1)			Single GPU (case 1)		
	Min (≥)	Exactly (=)	Max (≤)	Min (≥)	Exactly (=)	Max (≤)
10	16271.30	16270.40	19907.00	701.90	704.60	718.20
100	16274.30	16459.80	20805.30	696.50	677.30	697.40
1,000	16337.40	16375.20	19975.20	685.80	765.70	707.50
10,000	16304.00	16305.10	19923.00	686.90	707.50	723.50
100,000	16654.40	16653.00	20279.10	700.10	740.00	715.70
1,000,000	20941.40	20924.40	24566.80	925.70	928.00	969.90
10,000,000	117956.60	120007.50	124724.60	5549.70	5611.40	5641.90

Table 5.8: Role cardinality restrictions with a varying number of role assertions on case 2.

#Role Assertions	Single CPU thread (case 2)			Single GPU (case 2)		
	Min (≥)	Exactly (=)	Max (≤)	Min (≥)	Exactly (=)	Max (≤)
10	16270.20	17757.70	20808.40	699.40	757.20	726.00
100	16262.40	16268.20	19900.80	692.00	706.60	762.10

1,000	16327.50	17957.80	20022.40	697.50	691.20	705.00
10,000	16304.80	16296.80	20096.20	692.70	686.00	744.80
100,000	16504.20	16510.60	21094.50	763.10	778.00	810.80
1,000,000	18792.30	18792.10	22430.50	1373.80	1429.20	1448.20
10,000,000	44106.30	46097.80	51777.30	7459.10	7510.00	6389.10

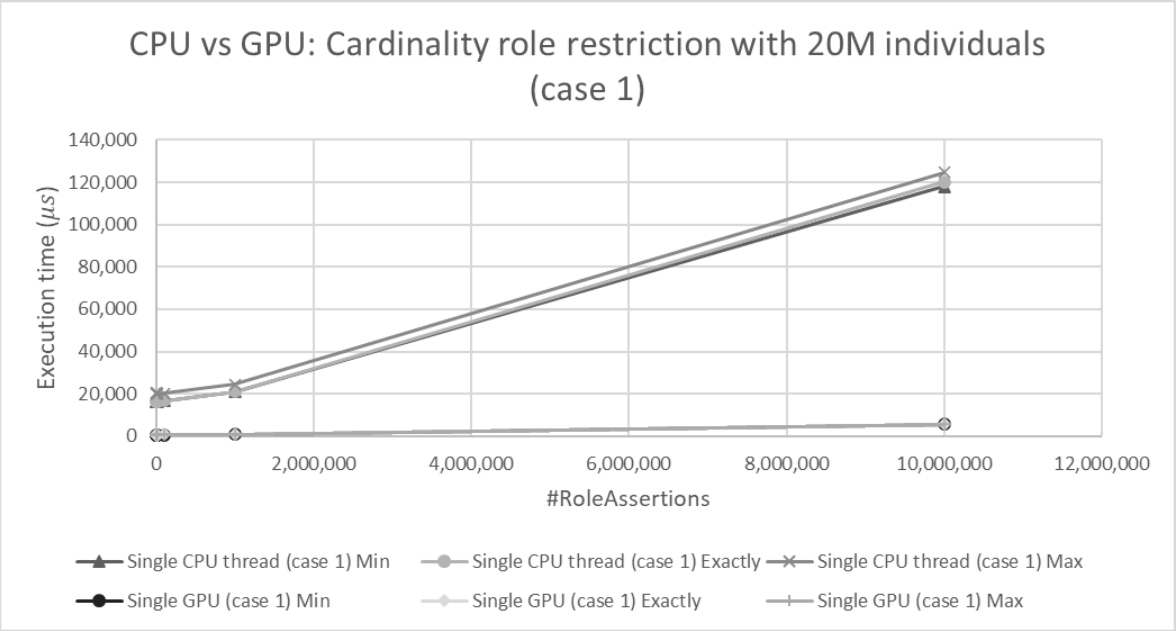


Figure 5.27: Plot of cardinality role restriction in case 1.

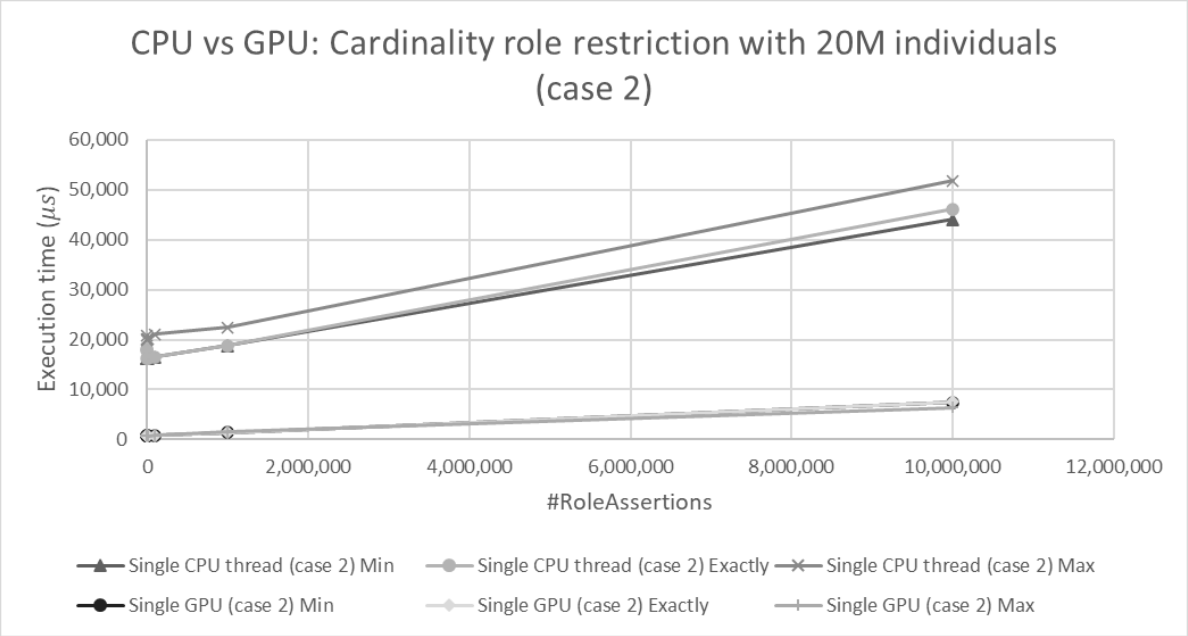


Figure 5.28: Plot of cardinality role restrictions in case 2.

For restrictions that share similar experimental results, their plot lines overlap and therefore may not be easily seen on the plot graph (e.g. MIN, EXACTLY, and MAX on the same number of assertions). From the experiments, the following observations are made:

- The GPU outperformed a single CPU execution in this restriction regardless of the number of assertions, similar to existential and universal restrictions.
- The role cardinality restriction exhibited similar behaviour to the existential restriction, by which case 1 is its best case and case 2 as its worst; on case 1, the GPU is 22 times faster on 10 million assertions (using the MAX cardinality restriction).

In terms of the three role restrictions, their performance gain (relative to single CPU) is in this order (from fastest to slowest): universal restriction (31 times faster), role cardinality restriction (22 times faster), and existential restrictions (18 times faster).

Concrete role restrictions

The evaluation of the concrete role restrictions follow a similar approach to that of the cardinality role restrictions. However, because of the algorithmic design of this operator, the values of assertions themselves have no impact on the execution time. Since concrete role assertions are represented in memory in a similar way to the class memberships matrix, an experiment with a varying number of individuals provides a sufficient evaluation of this operator's performance. See Table 5.9 for the experimental results and Figure 5.29 for its graph.

Table 5.9: Concrete role restriction with a varying number of concrete role assertions.

#Concrete Role Assertions/ individuals	Single CPU thread			Single GPU		
	Min (≥)	Exactly (=)	Max (≤)	Min (≥)	Exactly (=)	Max (≤)
10	0.00	0.00	0.00	47.80	50.90	48.00
100	0.00	0.00	0.00	41.20	42.00	41.90
1,000	0.00	0.00	0.00	41.10	43.90	41.80
10,000	5.00	6.00	5.10	41.50	42.70	42.70
100,000	57.10	63.20	57.10	42.60	42.90	42.60
1,000,000	580.30	674.00	599.50	58.10	59.00	58.20
10,000,000	4861.20	6464.80	4861.60	240.20	242.40	244.70

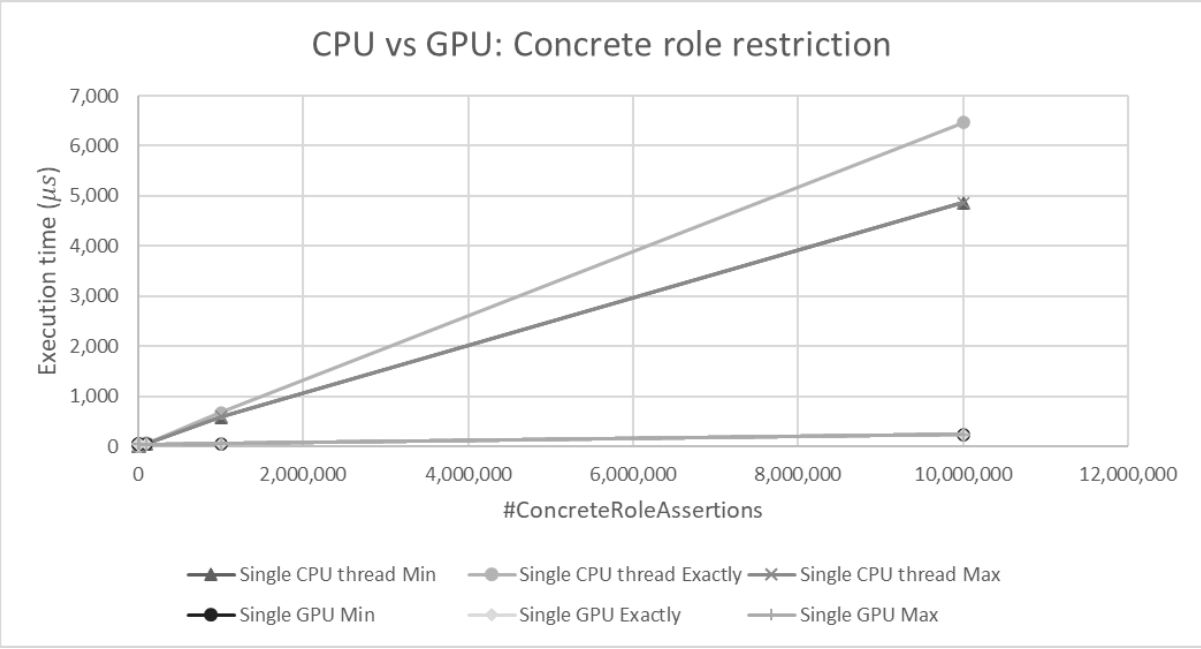


Figure 5.29: Plot of concrete role restriction with a varying number of concrete role assertions.

According to the experiments, the GPU outperformed a single CPU execution when the number of individuals was ≥ 1 million; that finding is similar to the conjunction and disjunction operations, even

though speedups starts on $\geq 100{,}000$ individuals on these two restrictions, yet becomes more pronounced on ≥ 1 million individuals. In terms of all DL operations, the GPU outperformed a single CPU on large datasets. For role restrictions, even though they used atomic operations, they still outperformed single CPU execution by a large margin. Moreover, the execution times for DL operations in the GPU grew linearly with data size. In the next section, we evaluate the performance for complete hypotheses in a single and remote GPU.

5.8.3 Evaluation of hypotheses in a single GPU

After evaluating individual DL operators, we evaluated the computations of full hypotheses (i.e. their individual DL operators, their coverage, and their score). Two experiments were used to study the performance of a single local (same machine) GPU and a remote GPU (including its additional overhead). The experiments consisted of executing a hypothesis (conjunction of four classes) on 1 million individuals; the choice of four classes is based on the assumption (by this thesis) for the average size of a hypothesis, where any of these four classes may be a simple class or an already computed DL operation (i.e. a complex class). The hypothesis was replicated, then evaluated on a local and a remote GPU in order to study the overhead impacts on the performance. The remote GPU server was running on the same machine as the client to study the communication overheads between the client and the server in isolation (i.e. in a perfect network). See Table 5.10 for the experimental results, where the execution times (and the overheads) are in milliseconds and where the average of 10 readings is used. A hypothesis evaluation of a single CPU is provided to establish the baseline performance.

Table 5.10: Experimental results on a single local versus remote GPU.

#Hypotheses	Single CPU	Local GPU	Remote GPU	Remote GPU overhead (computed)
1	5.00	1.10	2.00	0.9
10	51.40	16.70	18.30	1.6
100	537.20	171.50	194.50	23
1,000	5371.00	1719.90	1935.80	215.9
10,000	52063.40	17169.10	19361.00	2191.9

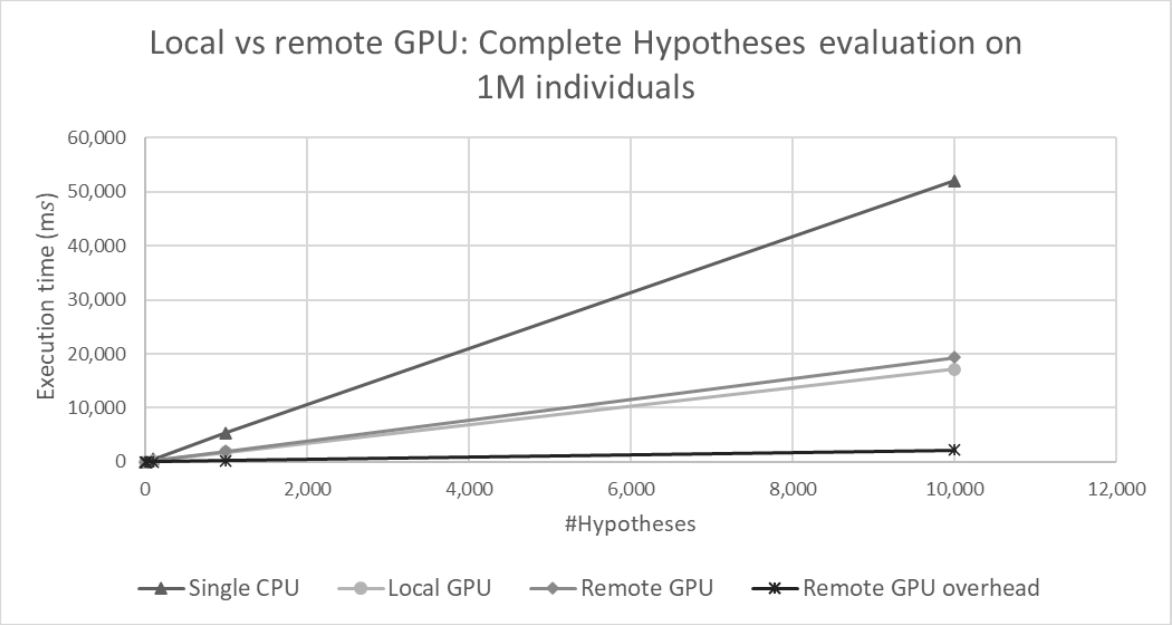


Figure 5.30: The plot of evaluating hypotheses on single local versus remote GPU.

Based on the experiment's results, the following observations were made:

- A single local GPU was faster than a single CPU evaluation – around 3 times faster for 10,000 hypotheses.
- Even when hypotheses are sent through a network for evaluation (with additional overheads), a remote GPU was always faster than a single CPU evaluation – around 2.7 times faster for 10,000 hypotheses.
- The remote evaluation overhead (serialization/deserialization and network transfer) were minimal (only around 2.2 seconds for 10,000 hypotheses).

To summarize the observations, a GPU is faster regardless of whether its connection is local or remote. All the conducted experiments so far used a single GPU (Nvidia GeForce RTX 2080). Next, we evaluate hypotheses on multiple GPUs – that is, through the complete multi-GPU engine.

5.8.4 Evaluation of hypotheses in multiple GPUs

In this section, we evaluate the complete multi-GPU engine, thus addressing the fourth hypothesis (Adding more parallel processors reduces ILP learning time) in the second research question (To what degree can parallel computing improve ILP learning in DL?). Two GPUs were used: Nvidia GeForce RTX 2080 (GPU1) and Intel UHD Graphics 630 (GPU2). Two variations of the multi-GPU engine were used. In the first variation, both GPUs were used directly (as local GPUs). In the second variation, the GPUs were treated as remote GPUs (to study their overheads). According to the performed experiments, when a GPU is used locally or remotely (including its overhead), it still outperforms a single CPU.

Multi-GPU scheduler evaluation

The multi-GPU scheduler was evaluated for both variations (local vs. remote) to see how GPU's scheduling priority is determined in both variations; see Table 5.11 for the experimental results of the

probing query on both multi-GPU variations. The probing query execution times (10-value average) are in microseconds. See Figure 5.31 for the GPUs' scheduling priority in both variations.

Table 5.11: Experimental results of the probing query on local versus remote multi-GPU engine.

#Individuals	Multi-GPU scheduling priority			
	All local GPUs		All remote GPUs	
	GPU1	GPU2	GPU1	GPU2
1	185.90	169.30	568.40	484.10
10	179.40	151.90	587.00	348.90
100	160.50	154.20	562.10	471.40
1,000	166.10	154.30	568.60	375.90
10,000	174.20	183.00	595.80	457.00
100,000	316.60	398.60	744.90	795.70
1,000,000	1862.10	2307.80	2258.20	2697.40
10,000,000	16290.80	23712.90	16675.60	24524.70

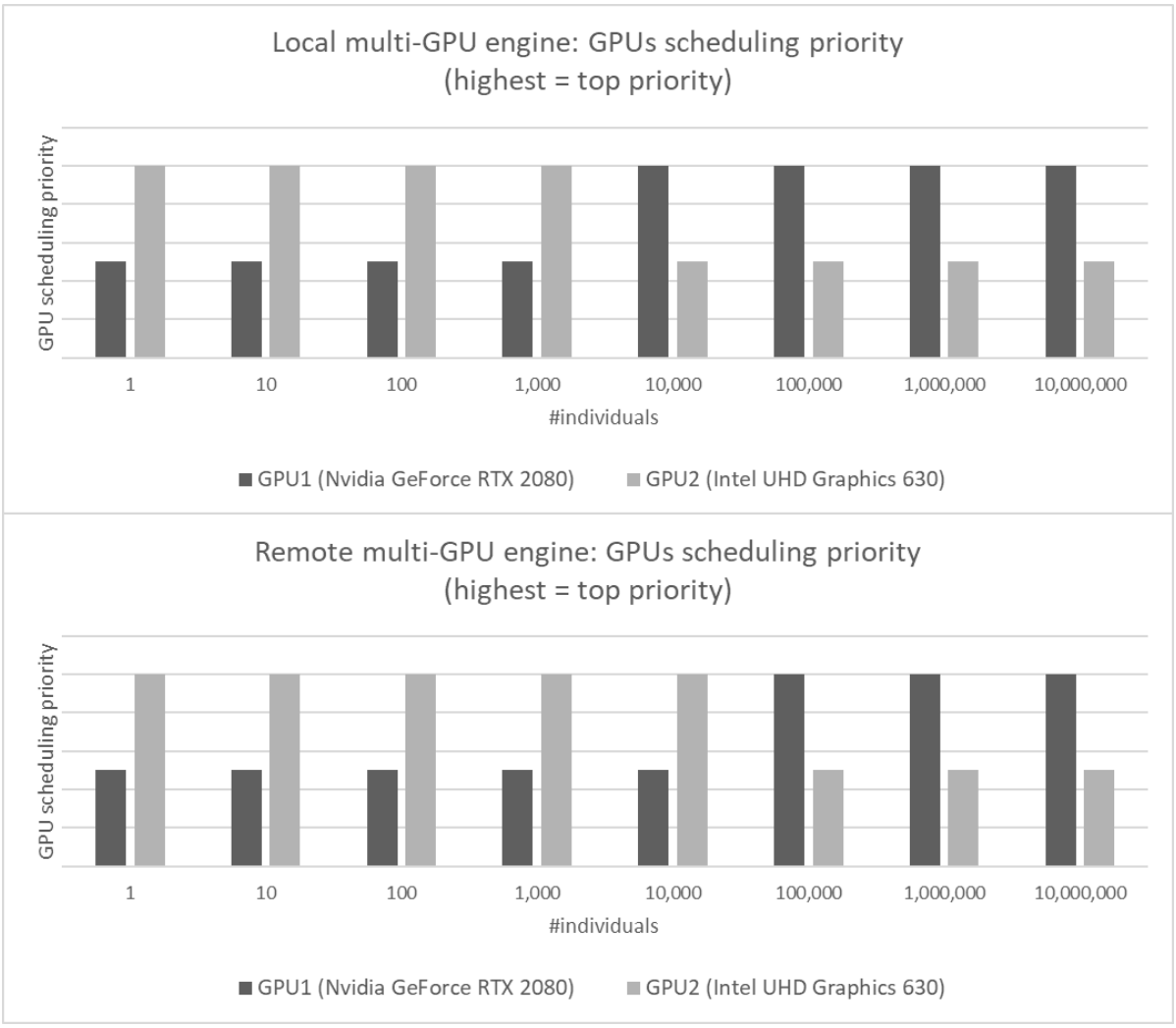


Figure 5.31: Plot of the GPU's scheduling priority on local and remote multi-GPU.

As discussed above, the probing query is used to roughly gauge the computing performance for each GPU. The aim of this experiment was to study how the complete multi-GPU engine prioritized which GPU to use first based its on probing query performance on a given dataset. According to experiment results, GPU2 was prioritized on smaller number of individuals than GPU1 since it executed the probing query faster on smaller datasets. However, when the number of individuals reached and exceeded 10,000, GPU1 was prioritized. For remote evaluation, GPU1 was prioritized when the number of individuals reached 100,000 because in this case, the probing query was affected also by network communication overheads. In conclusion, the multi-GPU scheduler demonstrated its adaptiveness by prioritizing GPUs according to the dataset size.

Local and remote multi-GPU evaluation

As a reminder, multiple GPUs were used to address the fourth research hypothesis. For both variations, a hypothesis (conjunction of four classes) was replicated and then evaluated on 1 million individuals. For the three scheduling strategies, experimental results are provided in Table 5.12 for the first variation (local GPUs) and in Table 5.13 for the second variation (remote GPUs). All execution times (10-value

average) were in milliseconds. See Figure 5.32 and Figure 5.33 for the graphs of the local and the remote multi-GPU engines, respectively. Also see Figure 5.34 for the activated GPUs by the (three) scheduling strategies on both variations.

Table 5.12: Experimental results on a local multi-GPU engine.

#hypotheses	Local multi-GPU engine on 1 million individuals		
	Gradient scheduling (1000 hyp/GPU)	Equal scheduling	Relative scheduling (GPU1 = 60%, GPU2 = 40%)
1	1.00	1.00	1.10
10	16.00	11.00	10.00
100	170.30	109.00	104.00
1,000	1728.80	1096.50	1036.40
2,000	2193.60	2195.30	2077.90
10,000	19755.20	10974.10	10374.80

Table 5.13: Experimental results on a remote multi-GPU engine.

#hypotheses	Remote multi-GPU engine on 1 million individuals		
	Gradient scheduling (1000 hyp/GPU)	Equal scheduling	Relative scheduling (GPU1 = 60%, GPU2 = 40%)
1	2.00	2.00	2.00
10	18.40	11.00	10.00
100	194.60	119.90	118.00
1,000	1946.90	1224.90	1189.80
2,000	2451.40	2434.10	2376.10
10,000	22152.70	12188.50	11834.30

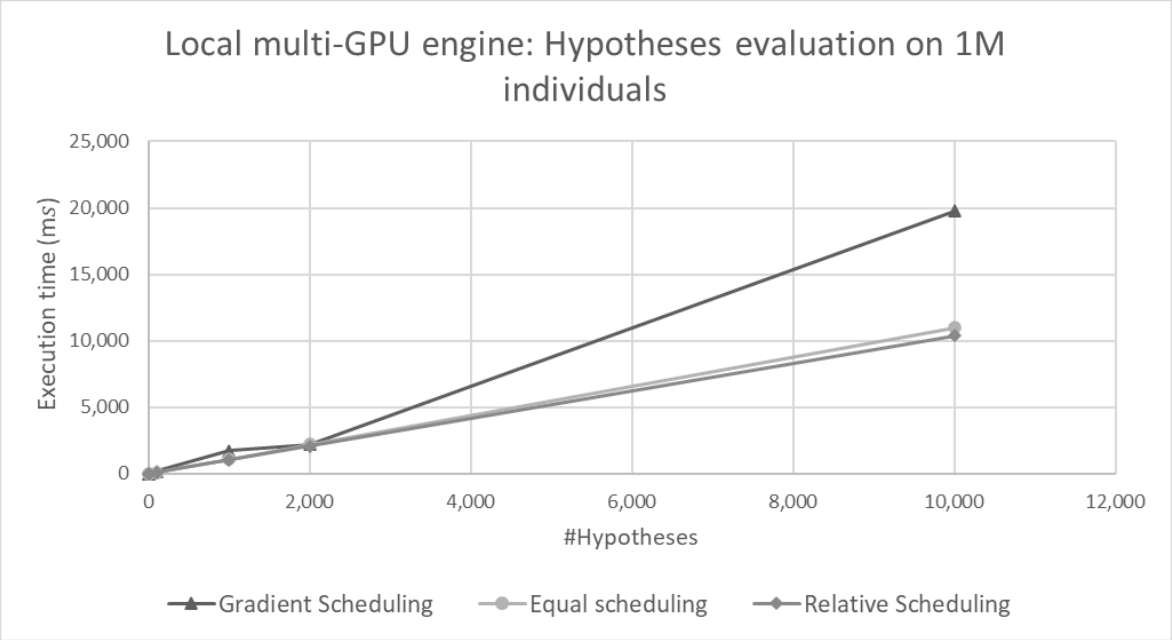


Figure 5.32: Plot of local multi-GPU engine.

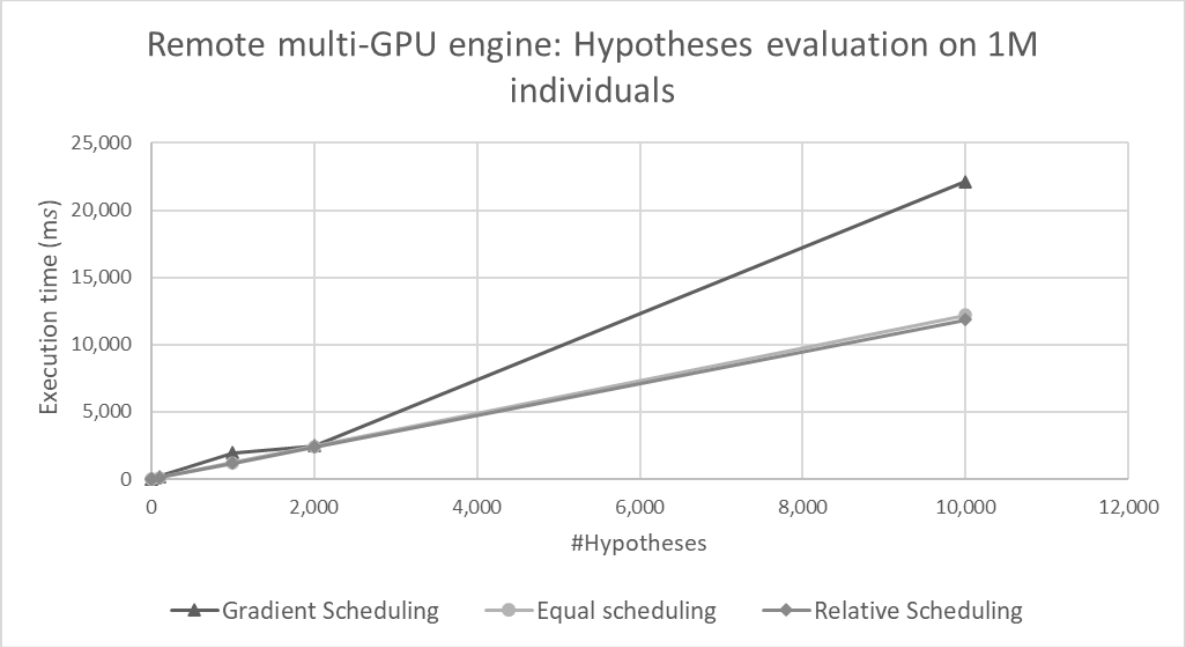


Figure 5.33: Plot of remote multi-GPU engine.

According to the experiments, the relative scheduling achieved the highest performance. This finding was expected since it follows the intuition that the powerful GPU is assigned a larger relative portion (percentage) of hypotheses instead of specifying a single constant portion number. It is worth noting that these scheduling strategies can be tuned further to achieve higher speeds.

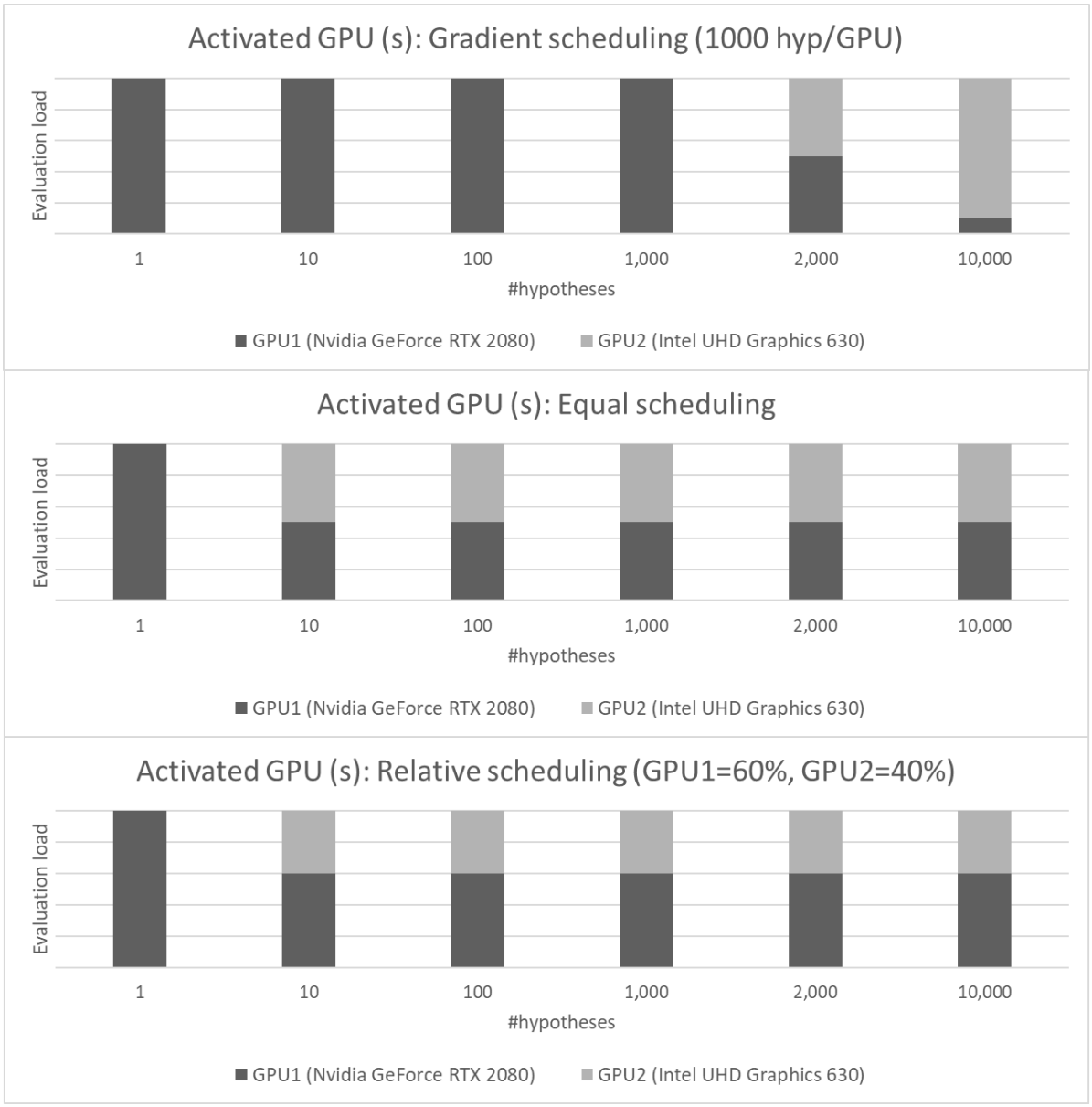


Figure 5.34: The activated GPUs in the three scheduling strategies for the local multi-GPU engine.

5.8.5 Discussion

The proposed multi-GPU engine was evaluated in a bottom-up manner, starting the evaluation from the small components and gradually moving up until the complete system was evaluated. Based on the experimental results, several observations (and trends) are apparent. First, in terms of the DL operators, the benefits of GPU-acceleration start to show when the number of individuals reaches 100,000 for conjunction/disjunction and 1 million for concrete role operators; for role operators, the speedups were observed from as low as 10 role assertions. Second, in terms of complete hypotheses evaluation (i.e. computing DL operators, computing coverage, and the scores) in a single GPU, the evaluation of hypotheses in a remote (network-linked) GPU had an acceptable level of (communication) overhead – that is, in the milliseconds range (up to 1000 hypotheses) instead of several seconds. Even with 10,000 hypotheses, the overhead (of around 2.2 seconds) is still reasonable. Thirdly, in terms of hypotheses

evaluation on multiple GPUs, the use of multiple GPUs for hypotheses evaluation resulted in speedups, with the right configuration of its scheduling strategy parameters. When 10,000 (identical) hypotheses were evaluated on 1 million individuals, the single local GPU (i.e. RTX 2080) took around 17 seconds to evaluate them. When multiple (two) non-identical GPUs were used, the evaluation took around 11 seconds for equal scheduling and around 10.4 seconds for relative scheduling. In other words, in a comparison with a single GPU evaluation, the performance of multi-GPU evaluation was around 1.6 times faster for equal scheduling and around 1.7 times faster for relative scheduling. However, for the gradient scheduling, the performance was actually worse (around 2.6 seconds longer) than a single GPU, which is due to poor configuration of the scheduling strategy. It is worth noting that the multi-GPU scheduler (based on its experimental results) managed to determine the right (scheduling) priority of a GPU based on its computing power on the given knowledge base. The reason for not including comparisons of a single GPU thread with a single CPU thread (for all the experiments) is that a single GPU thread resembles a single ALU, whereas a single CPU thread resembles a complete (single core) processor. The experiments were conducted on a multi-core CPU locked at its base clock (3.6 GHz on all cores) to ensure a consistent CPU performance (across all cores). The use of a single GPU for hypothesis evaluation outperformed the sequential (single CPU) evaluation by about 4.5 times on a single hypothesis. When evaluating 10,000 hypotheses on 1 million individuals, two GPUs were 5 times faster (using relative scheduling) than the sequential evaluation. The experimental results provide a clear piece of evidence that supports the first and fourth hypothesis. For the first hypothesis H_1 (Parallel computing reduces hypothesis evaluation time for large DL datasets), a single GPU can evaluate a hypothesis on a large DL dataset with a higher speed. For the fourth hypothesis H_4 (Adding more parallel processors reduces ILP learning time), adding a GPU improves performance even more – especially when a large number of hypotheses needs evaluation at the same time. The two addressed research hypotheses contribute partially to the two research questions: How can scalability issues of DL-based ILP learning be addressed using parallel computing (RQ1)? To what degree can parallel computing improve ILP learning in DL (RQ2)? In order to answer the two research questions, the rest of the research hypotheses are addressed in the next chapter:

- Hypothesis 2: Parallel computing increases the speed at which large search spaces can be explored.
- Hypothesis 3: Parallel computing improves ILP performance in general, even on smaller datasets.
- Hypothesis 5: Combining multiple types of processors (e.g. GPUs and CPUs) achieves further scalability improvements.

5.9 Summary

In this chapter, we propose a parallel multi-GPU engine that exploits the parallel computing capabilities of a single GPU or multiple GPUs to speed up DL hypotheses evaluation. We describe the proposed engine in terms of its knowledge representation, computational details of each (supported) individual DL operator, and the process of evaluating a single complete hypothesis. The hypothesis evaluation process is designed for a single local GPU and a single remote (network-linked) GPU. In addition, the hypothesis evaluation process has also been designed for multi-GPU evaluation, which includes the discussion of its multi-GPU scheduler (and its three scheduling strategies). Once the proposed engine was implemented, a set of experiments were conducted to evaluate its performance.

Based on the experimental results, the proposed multi-GPU engine proved its time efficiency in evaluating hypotheses against large (DL) knowledge bases, and it greatly outperformed the sequential evaluation (used by the DL-Learner, the state of the art) – especially the evaluation of a large number of hypotheses against a large knowledge base. For example, when evaluating 10,000 hypotheses on 1 million individuals, a single GPU is about 3 times faster than a single CPU; for multiple (two) GPUs, the evaluation is 5 times faster (using relative scheduling) than a single CPU.

In the next chapter, we explore how to incorporate the multi-GPU hypothesis evaluation engine into a newly proposed algorithm for ILP learning in DL inspired by the state of the art (i.e. OCEL) to produce a high-performance DL-based ILP learner.

Chapter 6: A scalable and parallel inductive learner

In this chapter, we combine the multi-GPU evaluation engine (discussed in the previous chapter) with the parallel hypotheses search (discussed thoroughly in this chapter). By combining the massively parallel hypothesis evaluation with the parallel exploration of the hypotheses space, a scalable and a parallel high-performance inductive learner in DL (for the $\mathcal{ALCQ}^{(\mathcal{D})}$ language) is achieved. We name the learner SPILDL (**S**calable and **P**arallel **I**nductive **L**earner in **DL**), a high-performance inductive learner capable of learning complex DL hypotheses from massive and complex DL datasets. This chapter starts with an introduction of the SPILDL architecture followed by its learning algorithms, implementation and performance evaluation. This chapter also answers the two research questions by addressing the following hypotheses:

RQ1: How can scalability issues of DL-based ILP learning be addressed using parallel computing?

H_1: Parallel computing reduces hypothesis evaluation time for large DL datasets

H_2: Parallel computing increases the speed at which large search spaces can be explored

H_3: Parallel computing improves ILP performance in general, even on smaller datasets

RQ2: To what degree can parallel computing improve ILP learning in DL?

H_4: Adding more parallel processors reduces ILP learning time

H_5: Combining multiple types of processors (e.g. GPUs and CPUs) achieves further scalability improvements

The first and fourth hypotheses are partially addressed in the previous chapter. By the end of this chapter, all research questions and their corresponding hypotheses are addressed.

6.1 Brief background on parallel search

According to the survey by (Fukunaga *et al.*, 2018), parallel best-first search algorithms can be classified into centralized and decentralized. In the centralized search, all parallel processors (or cores) share the same closed list and open list, this configuration has a potential bottleneck due to centralized data structures where parallel access to these structures is serialized in order to ensure data integrity, which may cancel out potential speedups. In the decentralized search, each parallel processor has its own open list and closed list, this configuration eliminates the potential bottleneck introduced by accessing a centralized data structure. However, decentralized search introduces some challenges such as the choice of load balancing strategy, reducing the communication overhead, and reducing the number of redundant search nodes (generated by other processors); checking the redundancy of a search node efficiently in a decentralized search, is a major challenge for parallel search algorithms. Although, some approaches exist for handling load balancing and the duplicate nodes detection using hash

functions. In the hash-based decentralized search, a specialized hash function is applied on a search node, the resulting hash is the address (or id) of a parallel processor that will handle that search node. Hash-based search, partitions (or assign ownership of) the search space among the parallel processors, where the nature of the partitioning is highly dependent on the characteristics of the hash function used, which consequently affects the learning performance.

The centralized and decentralized classifications are based on how the parallel search is performed. There are some parallel algorithms that address limited memory space, e.g. parallel memory-limited A*. In addition, there are some classifications based on the used technology such as parallel search using cloud environments. Some other classifications are based on the computing architecture used, e.g. CPU-based and GPU-based parallel search. The choice of a particular parallel search approach is dependent on the nature of the learning problem. For example, a centralized parallel search may work well with a particular search problem when the cost of expanding a search node is computationally expensive. However, when the cost of expanding a node is small, the centralized search may potentially introduce a lower performance than the traditional sequential search; here, a decentralized search may introduce the performance gains for that particular learning problem.

In the context of this thesis, we choose parallel beam search to accelerate OCEL's search algorithm, as we see such a parallel search approach is a suitable choice given OCEL's search characteristics.

6.2 SPILDL architecture

SPILDL builds upon the DL-Learner's OCEL learning algorithm and its refinement operator (discussed in chapter 3). It is worth noting, that SPILDL uses the full refinement operator of OCEL to learn a complete hypothesis (including a hypothesis made of disjunction of conjunctions) that describes all learning examples (identically to the OCEL algorithm). In other words, SPILDL is not restricted to only learning a single conjunctive hypothesis (i.e. a single conjunction of concepts) or a set of separate conjunctive hypotheses, where the final hypothesis is constructed (by the user) from these separate conjunctive hypotheses.

SPILDL uses a combination of data parallelism with task parallelism to accelerate key computations in the OCEL algorithm, where multi-core CPUs (task parallelism) are employed for parallel exploration of the hypothesis space and the multi-GPU engine (data parallelism) is used for parallel evaluation of hypotheses. See Figure 6.1 for the overall learning process in SPILDL, similar to that of the DL-Learner.

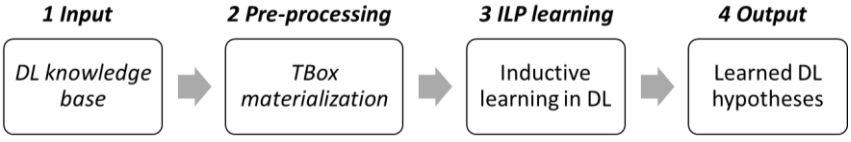


Figure 6.1: The ILP learning process in SPILDL.

The learning in SPILDL starts with an OWL ontology as the input. A DL reasoner then pre-processes the ontology (i.e. via TBox materialization) to prepare it for ILP learning. Thereafter, the SPILDL learning algorithm starts the (high-performance) inductive learning. Once the learning stops, the learned DL hypotheses are returned as the discovered solution. See Figure 6.2 for the architecture of the SPILDL learner.

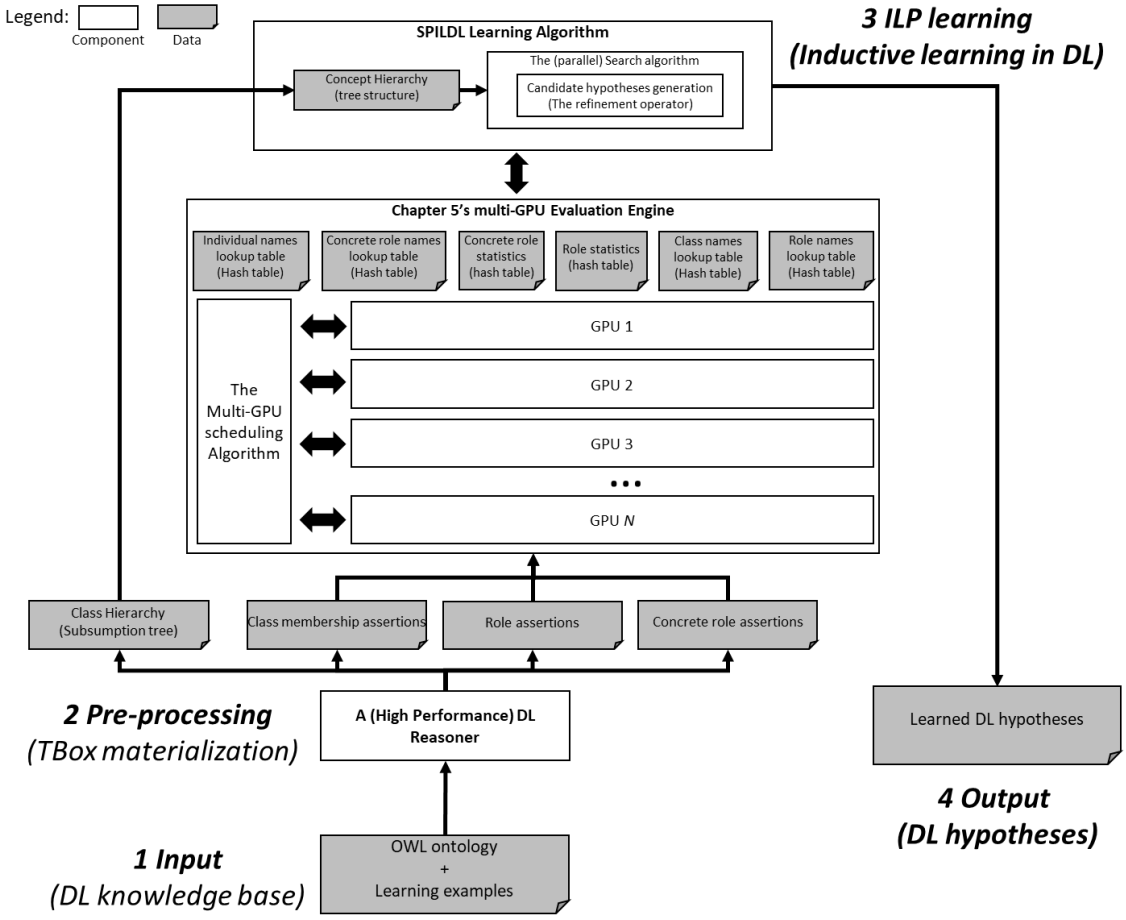


Figure 6.2: An overview of SPILDL architecture.

In SPILDL, both the DL knowledge base and the learning examples are contained in a single OWL ontology. The (OWL) ontology represents both the TBox and ABox, and the learning examples are represented as OWL individuals with class memberships to the positive and the negative classes, identical to the Knowledge representation discussed in chapter 5.

SPILDL pre-processes the ontology by passing it through a DL reasoner for TBox materialization. The output of the DL reasoner is an ontology that contains both asserted and inferred facts. In addition, the DL reasoner infers the class hierarchy, which is used by the refinement operator. Once the ontology is pre-processed (by the DL reasoner), the ontology is ready to be used for learning. The detailed design of the SPILDL learning algorithm is discussed below.

6.3 The parallel DL learning algorithm

The SPILDL learner builds upon the OCEL algorithm and its (extended) refinement operator. Since the OCEL's refinement operator uses information about the ontology (e.g. class hierarchy, roles, statistics), SPILDL computes this information (including the concrete roles' splits, as discussed in chapter 3) for the refinement operator from the (pre-processed) ontology. Once this is done, the actual learning starts.

The SPILDL algorithm uses OCEL with modifications to exploit the parallel computing capabilities for multi-core CPUs and GPUs. SPILDL learns hypotheses expressed in the $\mathcal{ALCQ}^{(\mathcal{D})}$ language. It uses multi-core CPUs for parallel exploration of the hypothesis space and GPUs (via the multi-GPU engine discussed in chapter 5) to accelerate hypotheses evaluation. As a consequence of this heavy use of parallel computing, SPILDL is able to handle massive and expressive knowledge bases.SPILDL uses parallel beam search to explore the best N nodes in the search tree in parallel, but with all nodes in memory. The search tree is represented as a 1D array of search nodes for a fast and parallel insertion of nodes into the search tree. The parallel search algorithms have inherent overheads that can potentially reduce their performance, such as the communication overhead, the search overhead, and load balancing. However, for SPILDL, the communication overhead is virtually non-existent for two reasons. First, because of the algorithmic nature of the parallel beam search, which expands all the best N nodes in parallel at a time, each thread expands its assigned node, and there is no communication between the threads. The second reason is that all the parallel expanding threads are managed and executed within a single (shared memory) machine, so that the cost of threads management and execution is negligible. SPILDL may have communication overhead only when its multi-GPU engine is using one or more remote (network connected) GPUs. In terms of search overheads, SPILDL has minimal search overhead because of its parallel reduction on the generated refinements. Since SPILDL reduces these two key communication overheads to the minimum levels, its parallel search performance is even higher.

For the redundancy checks, a hash table is used to provide fast tree node lookup. The learning in SPILDL starts by checking if the search tree contains node (s) worth expanding; if so, a (learning) loop iteration starts. In that loop, the best N nodes (in terms of their scores) are extracted from the search tree. Thereafter, a CPU thread is assigned (from a thread pool) to each individual node of the best N nodes. Similar to OCEL, each thread generates its node's refinements (in parallel with other threads) up to a certain length. Thereafter, the thread sorts its generated refinements into a certain deterministic order. In SPILDL, the sorting is done on the operands of the conjunction/disjunction based on their numeric IDs in their respective lookup tables and the types of these operands (e.g. atomic class name, conjunction, disjunction, role restriction, or concrete role restriction). After a thread finishes sorting its generated refinements, it then checks its redundancy against the search tree.

Once all threads finish generating, sorting, and then checking the redundancy of their refinements, a parallel reduction operation starts on the refinements of these threads to compute the final set of unique non-redundant nodes among these threads and the search tree. See Figure 6.3 for an example of this parallel reduction operation with 16 learning threads, where each box represents the refinements of a thread.

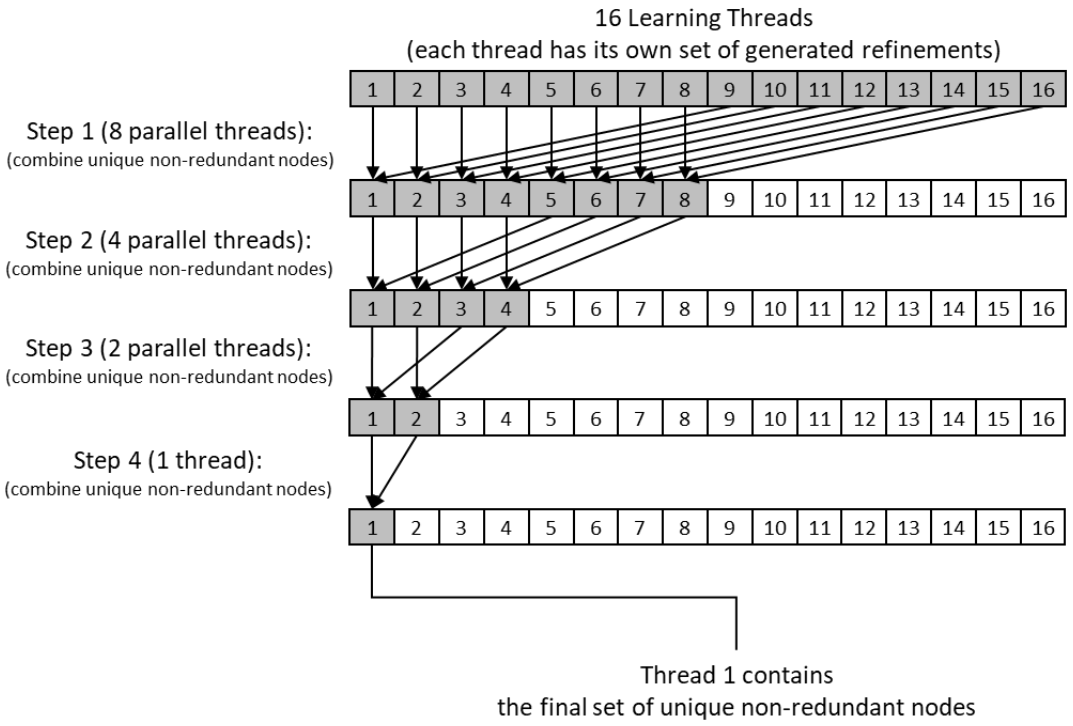


Figure 6.3: An example of the parallel reduction operation on 16 learning threads.

As seen in Figure 6.3, each thread carries out redundancy checks for its refinements against the search tree. In step 1 of the 16 threads example, eight threads start with each (parallel) thread comparing only two threads' refinements; the result of the comparison is a single set of unique and non-redundant refinements for the two threads. The process then continues with multiple parallel reduction threads, from 8 to 4 to 2 and finally to a single thread, which has the final set of unique and non-redundant refinements for all 16 threads. The design rationale for the parallel reduction operation is to address the issue of multiple parallel threads that generate similar (i.e. redundant) refinements.

Once the parallel reduction process is completed, the resulting refinements are added into the (redundancy) hash table. Thereafter, the refinements are sent to the multi-GPU engine (discussed in the previous chapter) for evaluation. Once all the refinements are evaluated, those within the acceptable noise level (i.e. $up > \lfloor noise \cdot |E| \rfloor$), as defined in OCEL (for handling noise), are added (in parallel) to the search tree; those beyond the acceptable noise level are discarded. After the search tree is updated with the new (good) nodes, a parallel sorting algorithm is then used to sort the tree nodes based on their

(OCEL) scores in descending order. This completes a single learning iteration. See Figure 6.4 for a flowchart of the SPILDL algorithm and Algorithm 6.1 for its pseudocode.

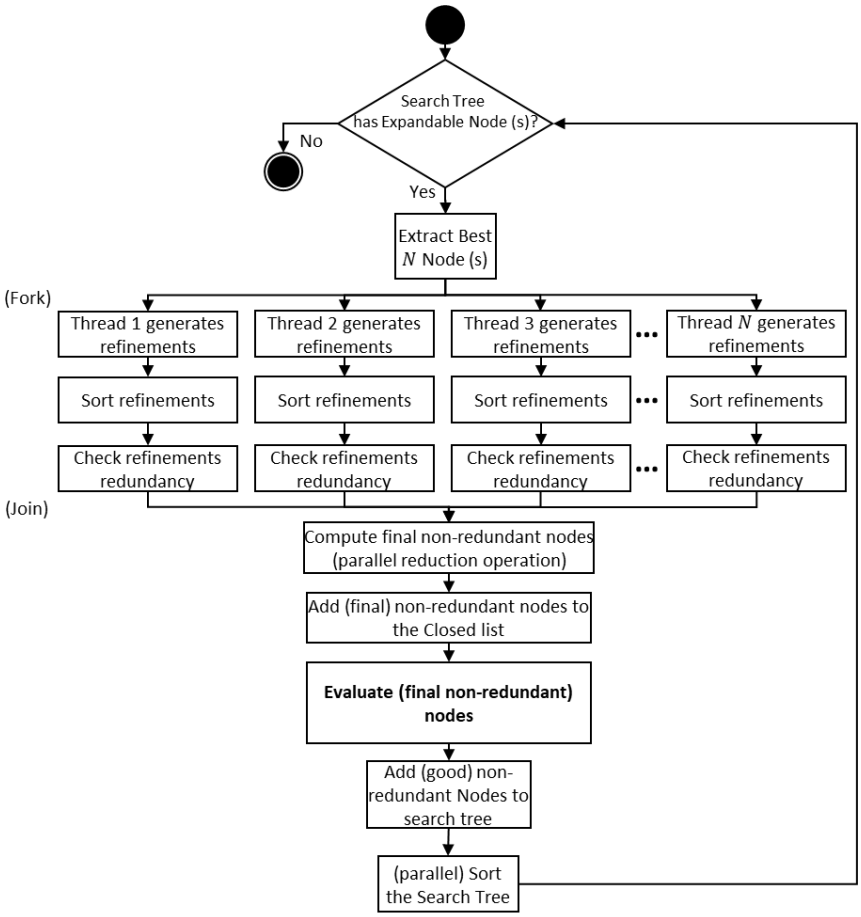


Figure 6.4: A flowchart for the SPILDL algorithm.

Algorithm 6.1: The pseudocode for the SPILDL algorithm.

The SPILDL learning algorithm

```
    Input: search tree array ST, the redundancy hash table RHT, beam width BW, number of
    final solutions limit
    Output: list of learned hypotheses LH
1
2   while ST has expandable nodes
3       N = extractBestNodes(ST,BW)
4
5       for every node n in N
6           fork expandSingleNode(n) //expand node, sort refinements, and check redundancy
7       end for
8
9       join threads //wait for all threads to finish
10
11      FinalRefs = computeFinalNonRedundantNodes(N) //parallel refinements reduction
12      Call AddRefinmentsToRedundancyTable(FinalRefs,RHT)
13
14      GoodRefs = ComputeEvaluationResults(FinalRefs)
15
16      Call addRefinementsToSearchTree(GoodRefs, ST)
17
18      Call parallelSortSearchTree(ST)
19  end while
20
21  LH = extractBestNodes(ST,limit) //return the best limit hypotheses
    Return LH
```

Parallel search algorithms have inherent overheads that can potentially reduce their performance, such as the communication overhead, the search overhead, and load balancing. However, for SPILDL, the communication overhead is virtually non-existent for two reasons. First, because of the algorithmic nature of the parallel beam search, which expands all the best *N* nodes in parallel at a time, each thread expands its assigned node, and there is no communication between the threads. The second reason is that all the parallel expanding threads are managed and executed within a single (shared memory) machine so that the cost of threads management and execution is negligible. SPILDL may have communication overhead only when its multi-GPU engine is using one or more remote (network connected) GPUs. In terms of search overheads, SPILDL has minimal search overhead because of its parallel reduction of the generated refinements. Since SPILDL reduces these two key communication overheads to the minimum levels, its parallel search performance is even higher.

6.4 Evaluation

The main purpose of SPILDL's parallel search is to speed up the process of hypothesis search – that is, finding solutions faster – especially for large and complex problems. In terms of experimental settings, the machine specifications are identical to those of chapter 5 (in Table 5.2). In this section, we focus on the evaluation of SPILDL's parallel search of a single GPU, although some experimental results are provided for the parallel search combined with the multi-GPU evaluation on large datasets.

6.4.1 Implementation

SPILDL was implemented using the C/C++ programming languages. To ensure that SPILDL correctly and accurately implements the OCEL algorithm, we used the DL-Learner source code (in Java)[19] to know crucial details regarding the OCEL algorithm, notably, its refinement operator and the procedure for computing a hypothesis length (including the length of each DL constructor); these crucial details for determining the length of a hypothesis (and a DL constructor) are only existent in the DL-Learner code, and the details for the refinement operator are clarified in the DL-Learner code. In other words, SPILDL does not assign arbitrary lengths (or weights) to DL constructors nor it follows a different procedure from OCEL for computing a hypothesis length. It is worth making such a distinction, because in OCEL, hypothesis length is one of its critical aspects that affects its learning performance (as a machine learning algorithm), by which it affects the score of each generated hypothesis and which hypotheses are allowed to be generated on a given hypothesis length; As a result, the order of which hypotheses are expanded next and the types of these hypotheses being expanded on each learning iteration are all affected, which will also reflect on the OCEL learning results (i.e. the quality of learned hypotheses as a machine learning models). The length of a DL hypothesis (including the length of each DL constructor) for OCEL, was already discussed in chapter 3 where these lengths are determined based on the DL-Learner source code.

[19] DL-Learner source code available at https://github.com/SmartDataAnalytics/DL-Learner/releases/tag/1.4.0

To implement multithreading in SPILDL, the OpenMP multithreading library was used to facilitate the construction of the parallel hypothesis search. For the materialization of the TBox, the reasoner Hermit is used for such a task, including the inference of the class hierarchy. Once the TBox is materialized, the (pre-processed) knowledge base could then be used for learning; the Hermit reasoner was not used further during the learning. Given the nature of multithreading applications and the nature of ILP learning, the use of object pool (a kind of software design pattern) was necessary to ensure that no performance bottleneck was introduced because of a large amount of expensive allocation/deallocation memory requests by the parallel threads.

6.4.2 Experiments' datasets

Several ILP datasets were used (including some classic ones), with a single CPU evaluation to establish a benchmark and performance baseline. Additional real-world datasets were used to test the performance of SPILDL. The classical ILP datasets are drawn from the DL-Learner's examples repository.[20] See Table 6.1 for brief statistics of the datasets used; the datasets highlighted in grey are real-world datasets that are not available in the DL-Learner's examples repository.

Table 6.1: Statistics for the used datasets.

Dataset Name	#Individuals	#Classes	#Class Assertions	#Roles	#Role Assertions	#Concrete Roles	#Concrete Role Assertions
Michalski Trains	50	12	113	5	149	0	0
Family (kinship)	31	5	67	3	95	0	0
Mutagenesis	14,145	88	14,375	5	26,533	6	7,044
Carcinogenesis	22,372	144	22,709	4	40,666	15	11,185
Moral reasoner	202	46	4,848	0	0	0	0
IMDB	1,224,835	29	3,255,390	1	3,431,489	1	388,269
Dunnhumby's retail	94,838	83	193,681	1	704,275	2	1,602

*Statistics include the positive and the negative examples assertions (and their two classes).

Michalski Trains Dataset

The Michalski's trains dataset is a well-known machine learning dataset that describes two types of trains in terms of their direction: eastbound and westbound. The main purpose of this dataset is to learn the description (or characteristics) of eastbound trains.

[20] Available at https://github.com/SmartDataAnalytics/DL-Learner/releases/tag/1.4.0

Family (Kinship) Dataset
This dataset describes the kinship relations among individuals. It can be used to learn the description of various relations, such as parent, father, mother, and grandmother.

Mutagenesis Dataset
This dataset contains data that describes chemical compounds, including information about atoms, chemical elements, chemical structures, and so on. It is used to learn the description of chemical compounds that can be used for mutagenesis (genetic mutation) prediction.

Carcinogenesis Dataset
This dataset similar to the Mutagenesis dataset in that it contains data about chemical compounds. However, this dataset is used to learn descriptions that predict carcinogenic compounds in rodents.

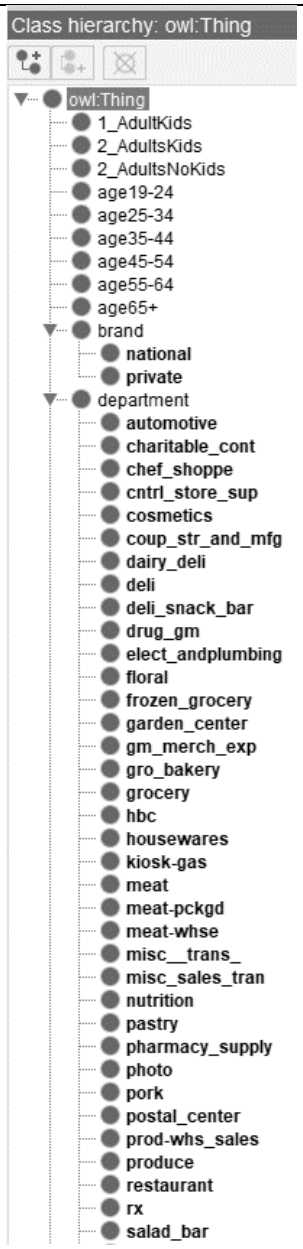
Moral Reasoning Dataset
This dataset describes the (rule based) reasoning process followed by a person about doing harm. This dataset can be used to learn the description of a guilty person (i.e. someone who did something harmful).

IMDB movies dataset
Internet movie database (or IMDB) is a dataset that describes movies and their related concepts, such as actors, directors, movie types, and movie roles. This relational dataset[21] was used to construct an OWL ontology, which was then used to learn the description of actors who acted in movies from 2000 and later; the positive examples are the set of these actors. The negative examples are the set of the remaining actors.

Dunnhumby's retail dataset
This is a retail dataset[22] that describes the transections of frequent 2500 customers at the level of their household. It contains details about the products sold and details about these households (in terms of their demographics). An OWL ontology was constructed from this dataset, which was then used to learn the description of the "25–34 years old" demographic.

For the classes, roles, and concrete roles of the two newly constructed ontologies and their hierarchies, see Figure 6.5.

[21] Available at https://relational.fit.cvut.cz/dataset/IMDb
[22] Available at https://www.kaggle.com/frtgnn/dunnhumby-the-complete-journey

Ontology	IMDB ontology	Dunnhumby's retail ontology
Classes		 . . .
Roles		

Concrete roles		

Figure 6.5: The two newly constructed OWL ontologies (snapshots from Protégé).

Table 6.2: The learning settings of the used datasets.

Dataset Name	#positive examples	#negative examples	Ignored classes
Michalski trains	5	5	-
Family (kinship)	5	12	-
Mutagenesis	13	217	-
Carcinogenesis	182	155	-
Moral reasoner	102	100	• "guilty" • "blameworthy" • "vicarious_blame"
IMDB	139,864	677,854	• "gender" • "genre"
Dunnhumby's retail	142	2,358	• "age19-24" • "age25-34" • "age35-44" • "age45-54" • "age55-64" • "age65+"

The learning settings for each dataset are shown in Table 6.2. The column for 'Ignored classes' tells the learning algorithm to ignore that set of classes when generating refinements; this exclusion reduces the search space; in other words, it restricts the hypothesis language. The noise tolerance for all datasets was set to 0% except for the "Carcinogenesis" dataset, where it was set to 30%. The number of splits for generating refinements for concrete roles was set to 12 for all datasets, and the maximum hypothesis length was set to 8. We limited the learning settings to only the ones described in Table 6.2.

In the following sections, we evaluated the learning algorithm on a single CPU, single GPU, and two GPUs. All the reported execution times are in seconds, and the values in parentheses with each dataset are its minimum acceptable OCEL accuracy – that is, the learning terminated for a dataset when a

hypothesis was found that satisfied its minimum OCEL accuracy. On each dataset, the beam width varied from one (sequential search) up to 16 parallel search threads; this limit was enforced based on the CPU used throughout the experiments (i.e. Intel i9), which has eight physical cores but can support up to 16 logical ones.

6.4.3 Evaluation of learning on a single CPU

In this section, we report our evaluation of classic ILP datasets and the two constructed real-world large datasets on a single CPU evaluation in order to establish the performance baselines. The experimental results are shown in Table 6.3; the performance baseline for the rest of the experiments is in the first row of the table (i.e. sequential search with a sequential CPU evaluation).

Table 6.3: Experimental results for learning with a single CPU.

Beam width/ #threads	**Learning a concept definition that satisfies minimum (OCEL's) accuracy on a single CPU**						
	Michalski's trains (100%)	**Family (100%)**	**Mutagenesis (46.9%)**	**Carcinogenesis (63.2%)**	**Moral reasoner (100%)**	**IMDB (94.2%)**	**Dunnhumby's retail (74%)**
1	0.01	0.00	0.13	0.28	69.23	1226.51	6.62
2	0.01	0.00	0.15	0.30	62.22	22.33	6.87
4	0.01	0.00	0.16	0.42	6.05	3.41	6.67
8	0.01	0.00	0.17	0.68	6.73	3.42	6.92
16	0.03	0.00	0.17	1.03	11.96	4.01	6.84

6.4.4 Evaluation of learning on a single GPU

In this section, we repeated the experiments in the previous section; however, a single GPU was used for evaluation instead of a single CPU. This variation was implemented to study the impact of a single GPU on learning performance. The experimental results are reported in Table 6.4.

Table 6.4: The experimental results for learning with a single GPU.

Beam width/ #threads	**Learning a concept definition that satisfies minimum (OCEL's) accuracy on a single GPU ("GeForce RTX 2080")**						
	Michalski's trains (100%)	**Family (100%)**	**Mutagenesis (46.9%)**	**Carcinogenesis (63.2%)**	**Moral reasoner (100%)**	**IMDB (94.2%)**	**Dunnhumby's retail (74%)**

1	0.02	0.00	0.13	0.27	69.34	454.89	6.47
2	0.02	0.00	0.14	0.27	62.56	3.61	6.63
4	0.02	0.00	0.17	0.37	6.17	1.04	6.50
8	0.04	0.00	0.18	0.60	7.18	0.74	6.63
16	0.07	0.00	0.17	0.89	12.08	0.96	6.89

6.4.5 Evaluation of learning on multiple GPUs

In this section, we repeated the experiments in the previous two sections. However, two GPUs were used for the hypothesis evaluation: GeForce RTX 2080 and Intel UHD Graphics 630.

Table 6.5: The experimental results for learning with two GPUs.

Beam width/ #threads	**Learning a concept definition that satisfies minimum (OCEL's) accuracy (two) GPUs: Using relative strategy (fastest GPU = 60% load, next fastest GPU = 40% load)**						
	Michalski's trains (100%)	**Family (100%)**	**Mutagenesis (46.9%)**	**Carcinogenesis (63.2%)**	**Moral reasoner (100%)**	**IMDB (94.2%)**	**Dunnhumby's retail (74%)**
1	0.01	0.00	0.14	0.29	70.07	394.03	4.52
2	0.01	0.00	0.15	0.28	64.58	2.20	4.64
4	0.02	0.00	0.15	0.39	6.64	1.39	4.65
8	0.03	0.00	0.17	1.03	7.22	1.39	4.67
16	0.05	0.00	0.15	1.06	12.11	1.81	4.79

6.4.6 Discussion

The main goal of the parallel search was to reduce the learning time through parallel exploration of the hypothesis space. For the Family dataset, the learning time for all experiments was less than 10 milliseconds (measured as 0.00 seconds) due to the simplicity of the learning problem. According to the experimental results, the parallel search did achieve speedups for some datasets, while on others, it did not significantly affect the learning time. The amount of speedup was highly affected by the nature of the dataset in terms of both structure and size. For example, in the IMDB dataset, two learning threads reduced the learning time from 1226.51 seconds to only 22.33 seconds – a speedup of around 55 times the original speed using single CPU evaluation; this is a massive performance gain. In the moral reasoner dataset, four learning threads reduced the learning time from 69.23 seconds to only 6.05

seconds – a speedup of around 11.4 times the original speed. On the other hand, in the Dunnhumby's retail and the Mutagenesis dataset, the parallel search had a negligible effect on the learning time. In terms of the learning threads, generally speaking, when 16 threads were used, no further speedups were achieved; this negligible effect was due (partially) to the CPU used in the experiments, which had eight physical cores. It is worth noting that the achieved speedups are due (in part) to the characteristics of beam search itself, i.e. speedups can be achieved by implementing a single core beam search. Although, parallelizing the beam search by expanding search nodes in parallel and checking their redundancy also in parallel (through parallel binary reduction), this will have a major performance increase for beam search - especially when large number of child search nodes are generated in a single learning iteration. The search effect of beam search can be clearly observed on the IMDB dataset experiments when the beam width is ≥ 2, i.e. when beam search is used.

In terms of hypothesis evaluation, the use of two GPUs instead of a single GPU introduced speedups only on the two larger datasets. For IMDB, the learning time was reduced from 454.89 seconds (single GPU) to 394.03 seconds (two GPUs) using a sequential search; this made the learning terminate roughly 1 minute sooner. On two threads, the learning terminated 1.4 seconds sooner. In the Dunnhumby's retail dataset, the overall learning terminated around 2 seconds sooner. However, the use of multiple GPUs made the learning on some datasets terminate roughly 1 second later, such as the Carcinogenesis dataset when using beam width of 8. The performance reduction by using multi-GPUs was due to the overhead of multi-GPU scheduling and due to the parameters (including the choice) of the GPU scheduling strategy. Given the configurability of the multi-GPU engine, further learning speedups can potentially be achieved through careful tuning of multi-GPU scheduling parameters.

The experiments were designed to study the combined effects of using parallel search and GPU-accelerated evaluation on accelerating hypothesis learning in DL. The multi-GPU experiments were specifically designed to study the existence of performance gains resulting from using multiple GPUs – that is, to determine whether multi-GPU hypothesis evaluation improves performance. However, the amount of performance gains from using multiple GPUs is highly affected by the multivariate nature of the multi-GPU engine, such as the types of used GPUs (are they identical?), the scheduling strategy, and its parameters, as discussed in the previous chapter. For a summary and visualization of the experimental results, see Table 6.6 and Figure 6.6. In Table 6.6, the results for parallel search with four threads are selected because they provide the best overall speedups across many datasets.

Table 6.6: A summary of the experimental results.

Search type	Sequential search			Parallel search using 4 threads		
Dataset	Single CPU evaluation (baseline)	Single GPU evaluation	Two GPUs evaluation	Single CPU evaluation	Single GPU evaluation	Two GPUs evaluation
Michalski's trains	0.01	0.02	0.01	0.01	0.02	0.02
Family	0	0	0	0	0	0
Mutagenesis	0.13	0.13	0.14	0.16	0.17	0.15
Carcinogenesis	0.28	0.27	0.29	0.42	0.37	0.39
Moral reasoner	69.23	69.34	70.07	6.05	6.17	6.64
IMDB	1226.51	454.89	394.03	3.41	1.04	1.39
Dunnhumby's retail	6.62	6.47	4.52	6.67	6.5	4.65

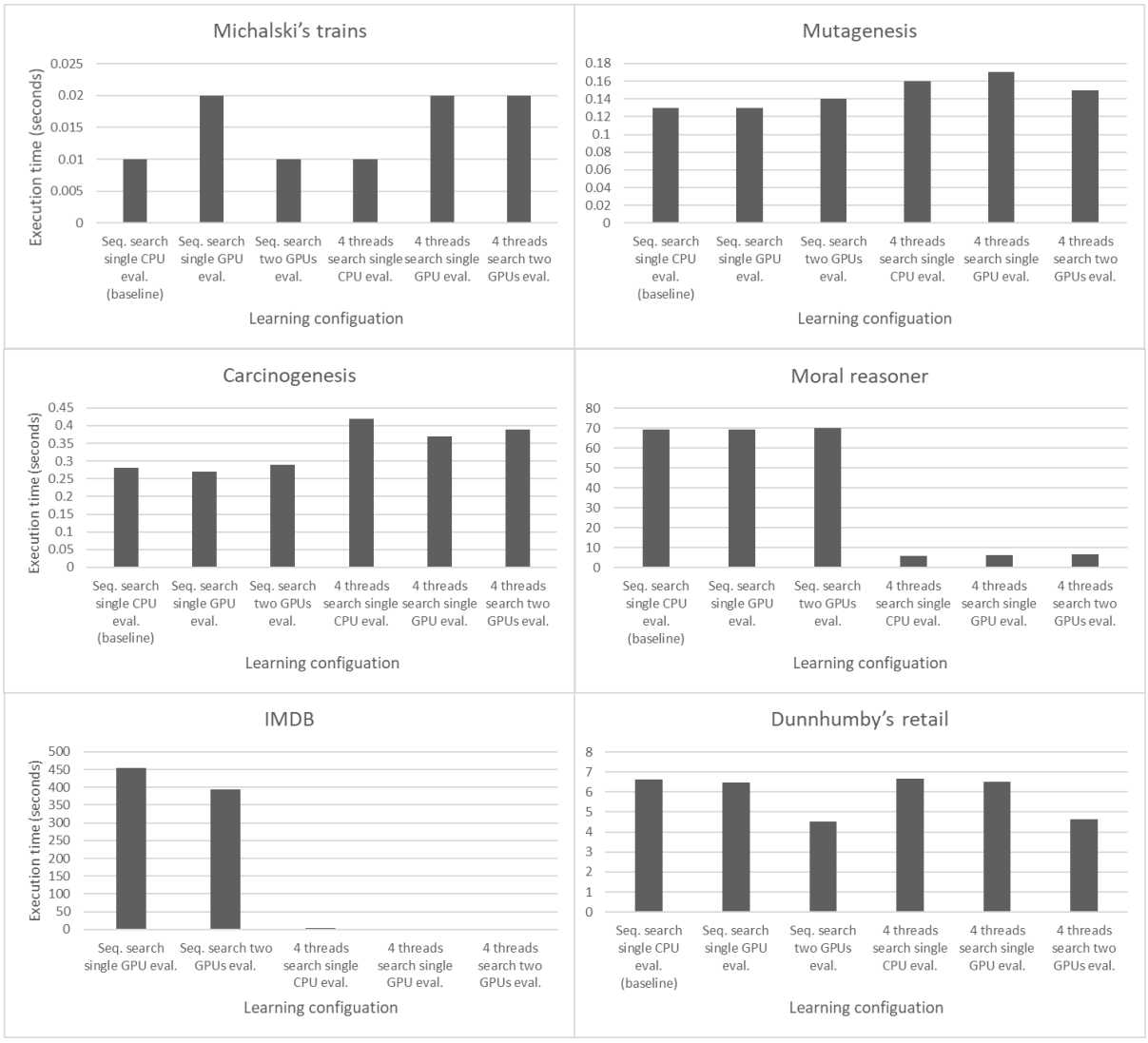


Figure 6.6**: A visualization of the experimental results.**

General patterns observed across all datasets are discussed below on the basis of hypothesis search and hypothesis evaluation. For hypothesis search, the search performance is not affected by the actual size of the dataset (i.e. the number of its ABox assertions) but rather by its search space, which is determined mainly by the number and layout (hierarchy) of its classes, roles, and concrete roles. Nevertheless, the ABox affects the search space to some degree by the number of concrete role splits (user-defined) and by the number of max role fillers, which are used by the refinement operator to generate concrete roles and cardinality roles refinements, respectively. For hypothesis evaluation, the dataset size is the only factor that affects the performance for hypothesis evaluation, regardless of the dataset layout, which only affects the required memory space.

This chapter completes the testing of all research hypotheses and consequently completes the answers to the two research questions. A summary of the testing of the research hypotheses is shown in Table 6.7.

Table 6.7: A summary of hypotheses testing.

Research hypotheses and questions	Hypothesis Scope		Hypothesis testing (based on experimental results)
	Chapter 5	Chapter 6	
H_1: Parallel computing reduces hypothesis evaluation time for large DL datasets	Yes	Yes	Chapter 5: Single CPU vs. GPU vs. two GPUs on synthetic data. Chapter 6: Single CPU vs. GPU vs. two GPUs on real-world data.
H_2: Parallel computing increases the speed at which large search spaces can be explored		Yes	Chapter 6: Sequential search vs. parallel search (with varying beam width) on multiple datasets.
H_3: Parallel computing improves ILP performance in general, even on smaller datasets		Yes	Chapter 6: Parallel search improves speed on small datasets with large search space, e.g. Moral reasoner.
H_4: Adding more parallel processors reduces ILP learning time	Yes	Yes	Chapter 5: Adding another GPU improves the performance of hypothesis evaluation, e.g. IMDB dataset. Chapter 6: Using parallel search reduces ILP learning time.
H_5: Combining multiple types of processors (e.g. GPUs and CPUs) achieves further scalability improvements		Yes	Chapter 6: Combining GPU-based hypothesis evaluation with multi-core (CPU) hypothesis search greatly improves scalability

6.5 Summary

In this chapter, we discuss our proposed scalable and parallel inductive learner in DL (SPILDL), which exploits parallel computing to accelerate both hypothesis exploration and evaluation. According to the experimental results, SPILDL demonstrated performance gains against a number of datasets, especially for large real-world datasets. These performance gains were observed at the level of hypothesis evaluation and in combination with the hypothesis (parallel) search. As a conclusion of this chapter, SPILDL achieved large speedups either by using parallel search or parallel evaluation separately or by combining them, which amplifies the performance gains (dramatically in some cases). In addition, if SPILDL did not improve performance in some cases (very small datasets), it still met the baseline

performance. In other words, the worst-case scenario for SPILDL is roughly the baseline performance (according to the experimental results).

In the next chapter, we describe a variation of SPILDL for mobile and embedded systems. The variation takes into consideration the nature of these systems in terms of their operating conditions and hardware capabilities. Parallel computing is recommended in general for improving computing performance. However, for embedded systems, the use of parallel computing may become an absolute necessity in order to facilitate an acceptable level of ILP learning performance. In addition, parallel computing improves the reliability of ILP learning results in embedded systems by employing the MISD parallel architecture to validate results' reliability while learning. This improvement is important since embedded systems usually operate in harsh conditions in terms of external factors, such as electric or electromagnetic noise, that may invalidate the integrity of learning results by altering memory contents or CPU clock speed. Addressing the two embedded ILP issues by parallel computing supports the answers to the two research questions.

Chapter 7: High-performance inductive learner in DL for embedded systems

In this chapter, we are concerned with the application area of ILP learning in DL – in particular, with high-performance ILP learning in DL for embedded systems. As a result, we introduce SPILDL-ES (**SPILDL** for **E**mbedded **S**ystems), which is a version of the main learner (i.e. SPILDL) optimized for embedded systems hardware. SPILDL-ES is a careful reengineering of SPILDL to achieve high-performance ILP learning in DL for embedded systems hardware. In comparison with SPILDL, SPILDL-ES is much lighter in terms of required memory space and computing power.

In addition to high performance ILP learning, SPILDL-ES monitors in real time the operating variables of the embedded system processors, such as the CPU/GPU temperature and the power supply condition (e.g. whether the power supply has sufficient and stable current). In addition, SPILDL-ES implements several measures from critical systems, such as the use of watchdog timers (also known as Computer Operating Properly, or COP timers) to ensure that the SPILDL-ES is operating properly. Watchdog timers are used in many embedded systems applications – especially in aerospace applications. Moreover, SPILDL-ES checks the quality of communication links to ensure that the remote two-way connection between the user and SPILDL is available.

SPILDL-ES tunes its performance in real time by processor throttling based on its operational variables. For example, when SPILDL-ES senses that one of its processors is overheating (because of heavy use), it throttles down its ILP learning performance to prevent hardware damage and to allow the hardware to cool down. The details of SPILDL-ES are described in the next sections. The main aim of this chapter is to address the high-performance DL-based ILP learning for embedded systems, an application area of ILP learning. By achieving the aim, this chapter augments the answers for the research questions and hypotheses as follows:

RQ1: How can scalability issues of DL-based ILP learning be addressed using parallel computing?

H_1: Parallel computing reduces hypothesis evaluation time for large DL datasets

H_3: Parallel computing improves ILP performance in general, even on smaller datasets

RQ2: To what degree can parallel computing improve ILP learning in DL?

H_4: Adding more parallel processors reduces ILP learning time

The chapter starts with a discussion of the architecture for SPILDL-ES. It then moves on to the discussion of its reliable ILP learning in terms of ensuring (results) data integrity. Thereafter, experiments for evaluating the learner are conducted, which also include their discussion. Finally, a hybrid learner is proposed that combines personal computers for hypothesis search with embedded

hardware for hypothesis evaluation; this hybrid learner can achieve scalable ILP learning in DL using relatively cheaper (embedded) hardware.

7.1 SPILDL-ES architecture

SPILDL-ES inherits most of the components of its original learner (i.e. SPILDL). However, SPILDL-ES has two additional components and modifications for its hypothesis evaluation component. The SPILDL-ES architecture can be seen in Figure 7.1.

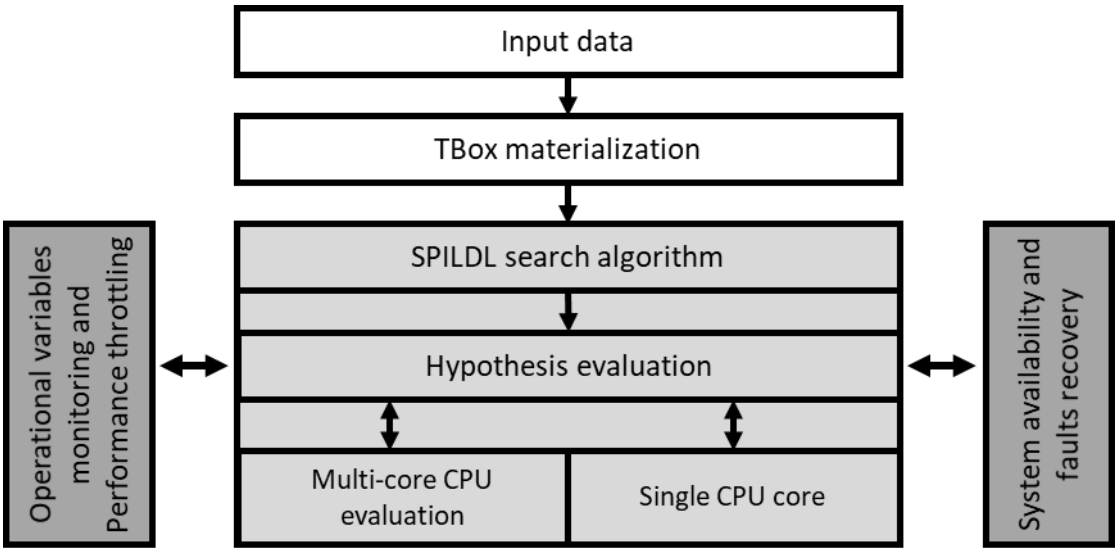


Figure 7.1: SPILDL-ES architecture.

In Figure 7.1, the original components of SPILDL (e.g. TBox materialization and SPILDL search algorithm) are clearly observed in plain colour. The light grey components are modified components of the main learner (SPILDL), and the dark grey components are the new components only available in SPILDL-ES. In SPILDL-ES, there are two variations of CPU-based hypothesis evaluation: single-core (sequential) and multi-core (parallel) evaluation, the latter of which uses multithreading on all CPU cores to extract as much performance as possible.

As mentioned earlier (at the beginning of the chapter), SPILDL-ES incorporates real-time monitoring of operational variables and techniques from critical systems, such as watchdog timers and MISD parallel architecture (for data integrity), to support reliable and stable ILP learning for embedded systems. The reason for incorporating such measures is that computers in embedded environments usually operate in harsh environments in terms of power supply (undervoltage/overvoltage, voltage spikes, noise, etc.); radio-frequency interference (RFI), which corrupts memory and processor clocks (if not protected appropriately, e.g. Faraday cage); varying operating temperatures/humidity; software and hardware bugs; and many other factors. The analysis and design of embedded systems – especially from a software engineering perspective – is a large and a complex topic. However, in order to stay within the scope of this work and this chapter in particular, only a few necessary aspects of embedded system design are considered here to support the discussion of this chapter – namely, high-performance

ILP learning in DL for embedded systems. Each specific SPILDL-ES component is discussed in the following sections.

7.1.1 Monitoring operational variables

This component of SPILDL-ES monitors its embedded hardware operational variables in real-time and throttles its learning performance accordingly. The monitoring and throttling are in action as soon as the ILP learning starts at the beginning of each iteration of the learning loop. At the beginning of each iteration, the power supply status and processor temperature are queried from the operating system. Once the operational variables are collected, the appropriate throttling policy is calculated, as shown in Table 7.1. Once the throttle policy is determined, it is applied in the next loop iteration.

Table 7.1: SPILDL-ES's throttling policy.

Operational variables (input)	**Throttling level (Output)**
Processor temperature	
≤ 49°C	Multi-core evaluation
50–60°C	Multi-core evaluation + few milliseconds delay
61–70°C	Single core CPU evaluation
71–75°C	Single core CPU evaluation + few milliseconds delay
≥ 76°C	Suspend the learning process until temperature drops to safe level
Undervoltage (Yes/No)	
No	[Keep same throttling policy]
Yes	Suspend the learning process until power is sufficient again

In Table 7.1, the processor temperature determines which throttle policy is activated. The undervoltage flag determines whether the embedded hardware is receiving sufficient power supply (i.e. $undervoltage = No$). If the processor temperature suggests a particular policy and undervoltage suggests another, the most performance-limiting policy of the two is selected. For example, if the board is operating at 65°C, it switches to single core evaluation; however, when the board encounters a low voltage (undervoltage = yes) caused by a low battery, for example, it temporarily suspends the learning until sufficient power is back. Next, we discuss the second new component of SPILDL-ES.

7.1.2 System availability and faults recovery

In many embedded systems, the hardware (typically an electronic board) is not easily accessible such as car engine computers, satellites, remote robots (e.g. Mars rover), and many others. Each of these applications has its own set of requirements and operating environments. However, across all these systems, an encountered hardware or software failure renders the embedded system inaccessible and attempting to gain physical access for repair or restart can be very expensive – especially for satellite systems. As a result, implementing measures to ensure high availability and appropriate fault recovery is a necessity for these systems, especially for systems that operate remotely in harsh environments (e.g. aerospace systems). In this component, several measures are implemented to ensure high availability and fault recovery for SPILDL-ES.

For high system availability, SPILDL-ES employs a hardware-based watchdog timer to ensure that any system freezes or crashes are recovered from by resetting the entire system. A watchdog timer is typically a hardware timer used to reset a computer when it reaches a predefined timeout. In other words, a watchdog timer assigns a predefined timeframe within which a task must be completed; otherwise the system restarts as a fault recovery mechanism. For example, consider task A, which takes roughly two seconds to complete, a watchdog timer is set to three seconds (as an example) and then activated before task A starts. After task A finishes its normal execution, it rests or disables the watchdog timer before it reaches its timeout to prevent the system from restarting. On the other hand, if task A is still executing and the watchdog timer limit is reached (i.e. three seconds has elapsed), a failure has occurred somewhere in task A. As a result, the watchdog timer resets the entire computer to recover from this failure, which takes (depending on the implementation) from few milliseconds to few seconds to completely boot up; for some embedded systems, the rest time may be as low as few microseconds for the entire system to come back online.

In many remote systems, a communication link has to be available at all times to ensure complete system access. In the context of this chapter, we assume the use of TCP/IP networks. In order to ensure that SPILDL-ES is always accessible to the user, every few seconds (e.g. five seconds) it sends a TCP packet, which is commonly known as a “heartbeat signal”. A heartbeat signal is a periodic message sent typically by a component or subsystem to indicate normal operation. Here, it is used to indicate communication reachability between SPILDL-ES and a remote user.

The two aforementioned measures help SPILDL-ES achieve high-performance ILP learning in addition to high availability and reliability. This level of performance and reliability is especially important for remote ILP learning in harsh operating environments (e.g. aerospace). In the next section, we discuss hypothesis evaluation in SPILDL-ES.

7.1.3 Embedded hypothesis evaluation

In many modern embedded systems, multi-core CPUs are typically used, and recently these processors have been augmented with SIMD instruction sets to further improve performance. Nevertheless, many accelerated computations by CPU-based SIMD are potentially memory bounded by a single CPU core (i.e. SIMD instructions of a single CPU core may overwhelm the memory bandwidth). In other words, sometimes combining SIMD instructions with multiple CPU cores may see no further performance gains because of the higher memory bandwidth requirement than what is available. In short, CPU-based (even multi-core) SIMD cannot replace GPUs because SIMD requires high memory bandwidth by design; that requirement is in addition to the data alignment requirement, which is relaxed on GPUs with which performance gains can be achieved regardless of data alignment (for GPUs).

Sometimes, embedded systems include GPUs that are typically used by the OpenGL-ES (OpenGL for Embedded Systems) API. For GPGPU in embedded systems, CUDA and OpenCL are in many cases not available, and the only approach to accelerate general-purpose computations using GPUs is through ComputeShader (part of OpenGL API). However, GPUs in embedded systems are mostly limited by design in terms of computing power and available GPU memory to necessary graphics tasks only. However, there is embedded hardware available that includes powerful CPUs and GPUs, such as the Jetson boards by Nvidia.[23] In the context of this chapter, SPILDL-ES is specifically designed for computationally limited embedded hardware. However, using SPILDL-ES with more powerful hardware results in more performance by default.

In SPILDL-ES, there are two CPU-based hypothesis evaluation engines: sequential and parallel. For sequential evaluation, a single CPU core is used to provide the baseline performance. For parallel hypothesis evaluation, multi-core CPUs are used to extract as much performance as possible. The advantage of using CPUs for hypothesis evaluation is the advantage of having virtually all the main memory available for storing all evaluation results and the learning dataset. In addition, no memory copies are needed before and after hypothesis evaluation, unlike GPGPUs where frequent data transfers between the main memory and the GPU memory are made to initialize the computation and retrieve the computed results.

The next section introduces the mechanism that SPILDL-ES employs to ensure reliability and the integrity of data in harsh operating environments, such as the existence of severe radio-frequency-interference (RFI), outer space, underwater, unstable power supply, and so on.

7.2 Reliable ILP learning (for data integrity)

In order for SPILDL-ES to ensure the reliability and integrity of results, it has an additional learning mode that uses multiple-instruction-single-data (MISD for short, commonly used in critical systems). In this learning mode, a single hypothesis is evaluated in parallel by four independent sequential CPU

[23] https://www.nvidia.com/en-us/autonomous-machines/

evaluation engines. The output from these engines (i.e. four results) is then compared to see whether they match. If they match, the results are reliable; otherwise, the evaluation for that hypothesis is restarted at most three times until reliable results are produced. If the three attempts fail, a hard reboot is issued to recover from the possibly corrupted memory (by external causes, such as RFI). A flowchart for critical hypothesis evaluation in SPILDL-ES can be seen in Figure 7.2.

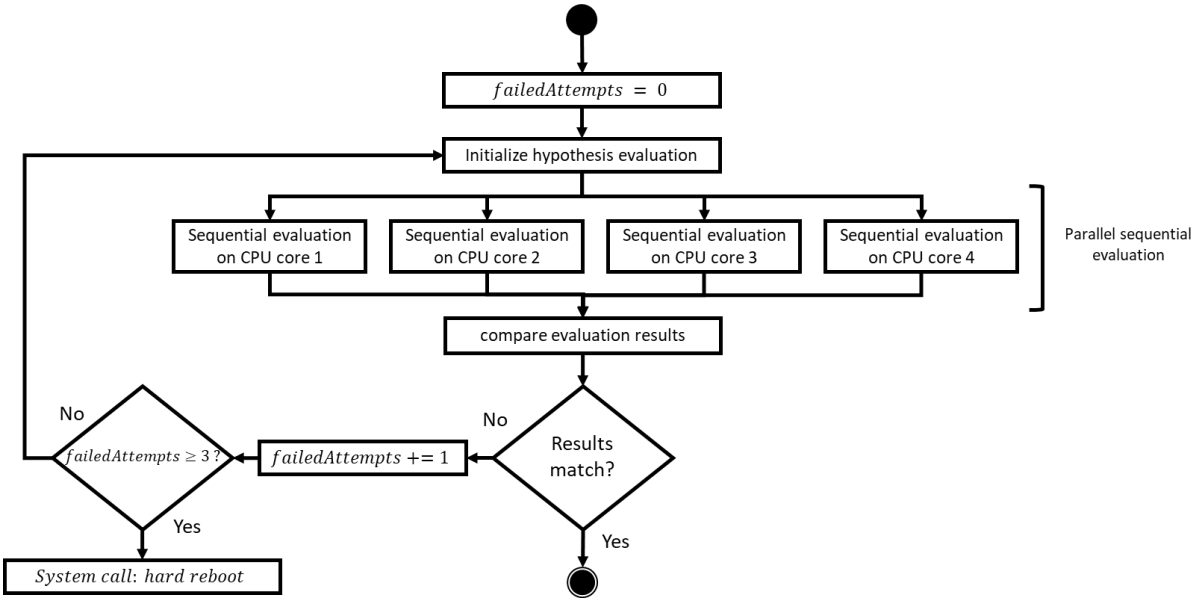


Figure 7.2: Flowchart for critical hypothesis evaluation in SPILDL-ES.

It is worth noting that each of the four sequential evaluation engines has an exact copy of the learning dataset; when critical learning mode is being used, the required memory space to store the learning dataset is four times larger than the sequential and the multi-core evaluations. However, since there are four copies of the dataset at different areas in memory, any severe electric noise (through the power supply) or RFI that affects the data integrity of memory contents is detected by SPILDL-ES when the four evaluation results are not matched, which indicates a potential data integrity issue.

Next, we conducted experimental results on the learning modes in SPILDL-ES, including sequential mode (baseline), parallel mode (parallel hypothesis search and/or evaluation), and critical mode.

7.3 Evaluation

In this section, we report our evaluation of the learning modes against the same seven datasets with the same configuration that was used in chapter 6. Since SPILDL-ES and SPILDL (the main learner) are both general-purpose ILP learners in DL, similar evaluation approaches were used.

7.3.1 Implementation

To facilitate the evaluations of the performance of the SPILDL-ES, its implementation was envisaged to sit on an embedded hardware such as Raspberry Pi,[24] a single-board computer with all the necessary

[24] https://www.raspberrypi.org/

hardware (CPU, main memory, communication protocols, etc.) in a single board. The main reason for choosing this board was due to its popularity and its low cost for its computing power. The Raspberry Pi configuration for SPILDL-ES is shown in Table 7.2, and the Raspberry Pi 4 board can be seen in Figure 7.3.

Table 7.2: Raspberry Pi configuration for SPILDL-ES.

Embedded hardware	Raspberry Pi 4 (latest model at the time of this work)
CPU	ARM Cortex-A72 – a quadcore 64-bit processor
Memory	8 GB
Operating system	Ubuntu MATE 64-bit (Linux-based OS)

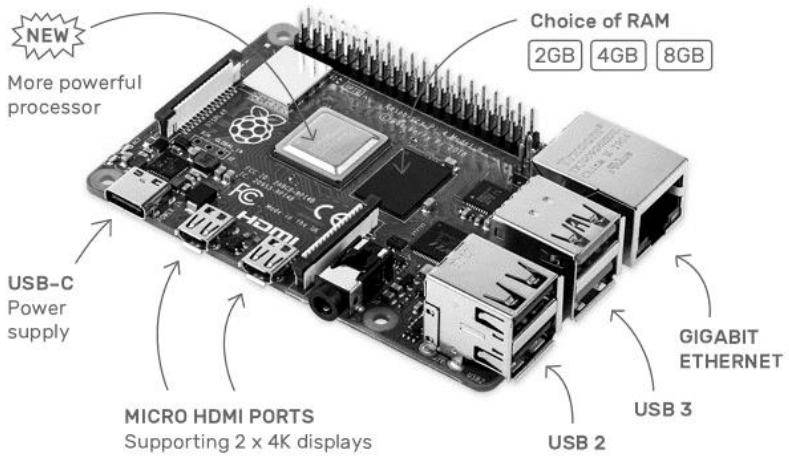


Figure 7.3: The Raspberry Pi 4 single board computer.[25]

SPILDL-ES is implemented in C/C++ and uses OpenMP API for multithreading (quad-core) hypothesis evaluation. The sequential search algorithm for SPILDL-ES is a serialized version of SPILDL (the original learner) parallel search algorithm with four parallel threads. The reason for using a serialized version of the parallel search was to reduce the memory space requirements and to improve learning speed in general. The reason for choosing the four threads for the parallel search was based conjunctively on the observations made in chapter 6 – namely, that four threads generally provide high performance gains across all tested datasets. In addition, four threads maximize the utilization of the Cortex-A72 quad-cores because four threads utilize all four cores, thus maximizing CPU usage efficiency; less thread count reduces CPU usage efficiency, and more threads do not necessarily yield more performance because of increased context switching overhead between threads (if thread count

[25] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

exceeds CPU core count) in addition to an amplified effect of CPU cache contention (if too many threads are used).

In terms of monitoring operational variables and computing throttle policy in (soft) real time, the learner reads its operational variables and updates its throttling policy at the beginning of each learning iteration; the learner also sends the periodic heartbeat signal (e.g. network ping). The watchdog timer is fully operational once SPILDL-ES (the program itself) starts; it is then reset every few seconds by hardware interrupt (to prevent hard reboot). The remote user interface for SPILDL-ES is shown in Figure 7.4.

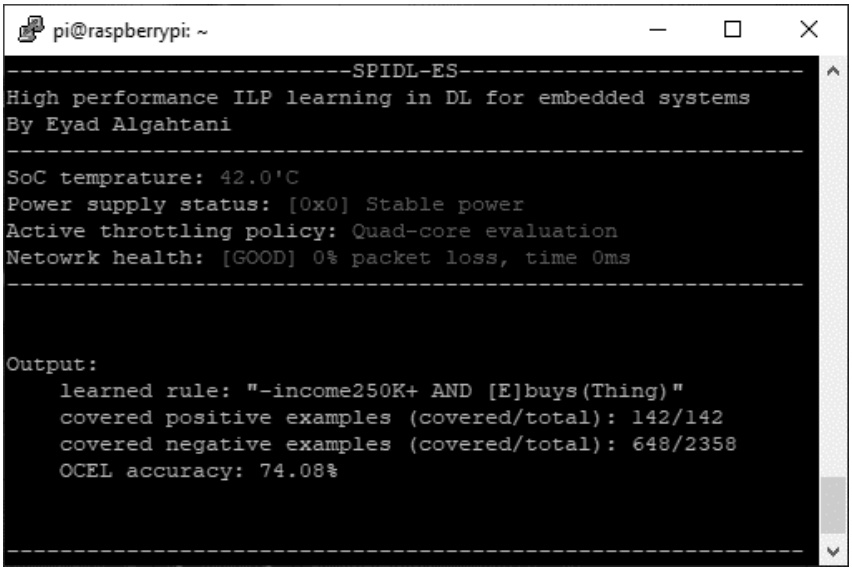


Figure 7.4: SPILDL-ES remote UI (through SSH protocol).

The reason for using a multi-core hypothesis evaluation for this chapter instead of SIMD-based evaluation (NEON for ARM CPUs) was based on the fact that SIMD in general requires data to be aligned in memory – that is, one memory transaction loads a set of data inputs, and one memory transaction stores its corresponding outputs. Data alignment for CPU-based SIMD is important (sometimes critical) to achieving performance gains. For GPUs, however, data alignment is recommended, though not critical, to achieve performance gains because GPUs (especially modern ones) can automatically handle data misalignments at the hardware level with greater efficiency.

7.3.2 Experiments

For the experimental results, the raspberry Pi 4 processor was de-clocked to 600 MHz from its original 1.5 GHz to simulate more limited processors and also to ensure that no performance achieved with SPILDL-ES was due to the use of higher clock speeds. The experimental results are shown in Table 7.3 for the three learning modes of SPILDL-ES, with execution times in seconds.

Table 7.3: SPILDL-ES experimental results.

Learning a concept definition that satisfies minimum (OCEL's) accuracy on the Raspberry Pi 4 (CPU de-clocked to 600 MHz from 1.5 GHz)					
Dataset	Learning mode				
	Sequential search and sequential evaluation (baseline)	Sequential search and quad-core evaluation	Quad-core search and sequential evaluation	Quad-core search and quad-core evaluation	Critical mode (quad-core search and parallel sequential evaluation)
Michalski's trains	0.15	0.15	0.13	0.13	0.13
Family	0.08	0.08	0.09	0.08	0.08
Mutagenesis	1.75	1.72	1.15	1.09	1.65
Carcinogenesis	8.89	8.57	4.21	3.89	7.96
Moral reasoner	152.31	151.78	50.51	50.34	50.89
IMDB	Could not load dataset (not enough RAM)				
Dunnhumby's retail	76.54	64.75	75.55	63.64	250.54

It is worth noting that for all experimental results, the watchdog timer was fully operational and real-time monitoring of operational variables was enabled. In addition, the periodic heartbeat signal (ping) was also working while the learning process was carried out. However, the throttling was disabled in order to accurately measure the learning performance based on different configurations of hypothesis search and evaluation approaches. The computation and implementation of throttle policy has virtually no overhead because it uses few if-statements to determine which policy to activate and because the activation of a particular policy is as simple setting the (array) index of which evaluation engine to use next.

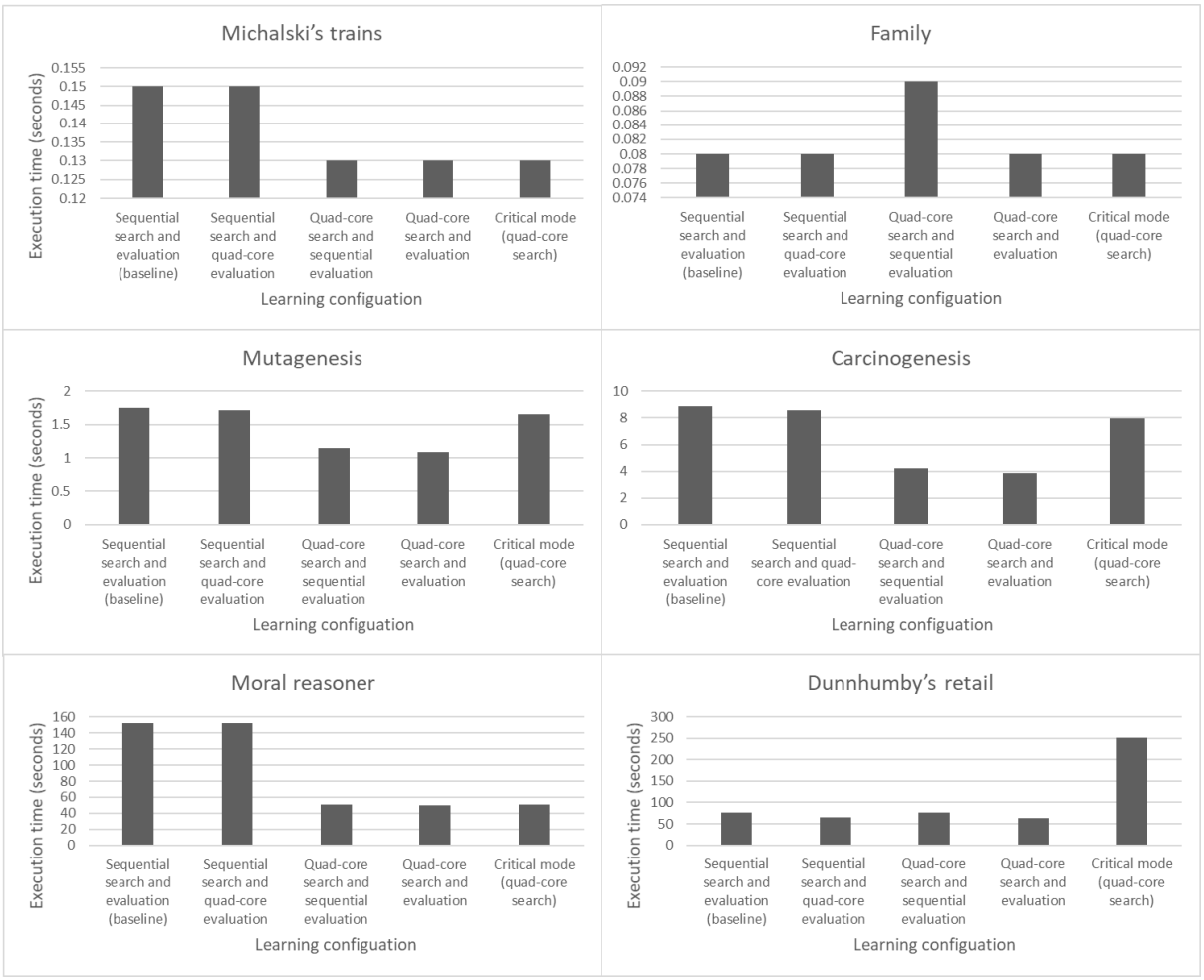


Figure 7.5: A visualization of SPILDL-ES experimental results.

7.3.3 Discussion

Based on the experimental results in Table 7.3, the following observations are made. First, for datasets with a small number of assertions and some with large search space (i.e. from "Michalski's trains" to "Mutagenesis"), SPILDL-ES seemed to meet the baseline performance across all parallel approaches. Second, for datasets with large search space (i.e. "Moral reasoner") and with a small number of assertions, the parallel hypothesis evaluation had negligible performance gains. However, parallelizing the search (even with sequential evaluation) reduced the learning time for the "Moral reasoner" dataset from 152.31 to 50.51 seconds, a threefold performance increase. For the "Dunnhumby's retail", parallel search had a negligible effect on performance regardless of the evaluation approach (sequential or parallel) due to the nature of the search space for this particular dataset. Although, parallel evaluation introduced 15.4% performance increase (from 76.54 down to 64.75 seconds) regardless of the search approach (sequential or parallel). The "Dunnhumby's retail" dataset is the second largest dataset in terms of its ABox assertions (after the "IMDB" dataset), and since parallel evaluation focuses on evaluating hypotheses against ABox assertions only, parallel evaluation will always achieve speedups when the ABox is large enough, based on the experimental results for the "Dunnhumby's retail" dataset in this chapter, and "IMDB" and "Dunnhumby's retail" datasets in chapter 6. Nevertheless, a general

pattern was observed on all experimental results; that is, the learner provided either a better performance (best case) or the baseline performance in the worst-case (apart from the critical mode).

For the critical (learning) mode (in the last row of Table 7.3), the experimental results are provided for the reliable ILP learning in DL, as discussed in section 7.2. The difference between the critical mode and other learning modes lies in their hypothesis evaluation; the hypothesis evaluation for the critical mode is parallelized not for performance but rather for reasons relating to data integrity and results reliability to ensure that the evaluation results are not affected by either internal (software issues) or external (electric noise, RFI, etc.) faults. Across all the datasets, the critical mode generally provided a better performance than the baseline, mainly attributed to the use of parallel search.

The reason for choosing parallel search for the critical mode was to reduce the overhead by the data reliability and integrity insurance. The critical learning mode has a key limitation (in its current design) caused by the parallel threads that evaluate the same set of hypotheses. The parallel threads compete with one another over CPU cache, introducing a major performance bottleneck – especially for large datasets. This bottleneck is known as false-sharing, and its effect can be clearly observed from the critical mode experiment for the "Dunnhumby's retail" dataset. Because the critical mode is not designed for performance but rather for ensuring reliability and integrity of learning results, it is most appropriate for learning applications where result reliability is more important than the learning speed.

SPILDL-ES addresses the same research hypotheses as SPILDL in an augmented way since both learners are used for general-purpose ILP learning in DL, where the core difference between the two lies in hypothesis evaluation: SPILDL-ES is designed for embedded systems architecture, and SPILDL is designed for the architecture of personal computers. However, SPILDL-ES employs additional types of parallel computing (e.g. MISD parallel architecture) to ensure data reliability and integrity.

7.4 Summary

In this chapter, we discuss SPILDL-ES, a variation of the main learner (described in chapter 6) targeting ILP learning in DL for the embedded systems domain. SPILDL-ES specifically exploits the limited available computing resources of embedded hardware in order to achieve as much learning performance as possible. We also introduce several measures, which the embedded learner (SPILDL-ES) can monitor and adapt to its potentially harsh (embedded) operating environment in real time, such as in satellite applications, which by their nature endure a large amount of cosmic and electromagnetic radiation. The learner also employs the use of heartbeat signals and a watchdog timer to ensure the reliability of the complete system, in which any failure results in a hard boot as a fault recovery mechanism.

According to the experimental results, the proposed learner in combination with all of its features enabled (except computing throttle policy) either produced a better learning performance or at least

maintained the baseline performance. In addition, the integrated measures for handling operational variables and the two reliability measures (heartbeat and watchdog) had insignificant overhead for the ILP learning process.

Chapter 8: Conclusion and future work

8.1 Conclusion

Parallel computing has (already) been used to accelerate many classical (Horn clause based) ILP learners. However, the acceleration of DL-based learners is non-existent even though DL is a sufficiently expressive knowledge representation formalism that has been used in many real-world applications to capture large and complex knowledge. Moreover, ILP learning in general suffers from many scalability issues because of its complexity in multi-relational learning. Consequently, the application of ILP learning in real-world scenarios is limited to small (but complex) datasets. Addressing the issue of scalability with ILP learners – especially DL-based ones – will open more frontiers for ILP as a large-scale machine learning technique.

The aim of this work is to address the scalability issues with ILP learning in DL by proposing a scalable and parallel inductive learner in DL (SPILDL) which combines the OCEL learning algorithm from DL-Learner (the state of the art) with intensive use of parallel computing. The chapters of this work contribute collectively to answering the two research questions: How can scalability issues of DL-based ILP learning be addressed using parallel computing (RQ1)? To what degree can parallel computing improve ILP learning in DL (RQ2)?

In order to answer the two research questions, we first investigated the factors within the parallel computing approaches and techniques that affect the scalability (and performance) of ILP learners (i.e. hypothesis evaluation and hypothesis search). We also reviewed the non-parallel approaches to improve ILP-related computations, such as query packs and sampling techniques, in order to provide a comprehensive review of other related factors to ILP scalability issues. A comprehensive analysis of the relevant literature and of the OCEL algorithm for areas that parallel computing can improve was also carried out.

For parallel hypothesis evaluation, we proposed and evaluated a multi-GPU engine for evaluating ILP hypotheses in chapter 5. The proposed evaluation engine uses a heterogeneous mix of local or remote (network-linked) GPUs and CPUs to collectively accelerate hypothesis evaluation. According to the experimental results, the use of a single GPU improves the performance for a single hypothesis evaluation by around 4.5 times for a local GPU (same machine) and by around 2.5 times for a remote GPU. By improving the performance for a group of hypotheses, multiple GPUs can be used to split the evaluation load. In this regard, two GPUs are around 1.7x faster than a single GPU for evaluating 10 hypotheses.

For parallel hypothesis search, we proposed and evaluated a parallelized version of the OCEL algorithm, one which uses parallel beam search but keeps all nodes in memory. The proposed parallel algorithm employs parallel reduction on the parallelly generated hypotheses to compute the final set of

unique and non-redundant search nodes from a closed list. According to the experimental results, performance improvement for hypothesis search up to 360 times faster than the original speed can be achieved. The performance for parallel hypothesis search is highly affected by the search space of the ILP dataset. For some datasets with a large search space, extremely high speedups are achieved, whereas in others with a small search space, negligible speedup is achieved. However, for parallel hypothesis evaluation, a consistent speedup is always achieved when the ILP dataset is large enough, regardless of its search space complexity.

Since the proposed learner SPILDL exploits the computing capabilities of several parallel processing architectures (multi-core CPUs and GPUs) at multiple levels, SPILDL is not only scalable with high-performance but also efficient in terms of using all available computing power to improve ILP computations. A summary of the results for the SPILDL strategies that affect the ILP learning process in DL can be seen in Table 8.1.

Table 8.1: A summary for SPILDL factors and their impact on ILP learning in DL.

ILP learning component	**Employed parallel approach**	**Parallel approach precondition**	**Particular area for performance improvement**	**Best case scenario**	**Worst case scenario**	**Considerations**
Hypothesis evaluation	Single GPU	Large dataset (in terms of number of assertions)	Evaluating a single hypothesis	4.5 times faster than sequential (CPU) evaluation	none	All datasets need to be in GPU memory
	Multiple GPU		Evaluating a group of hypotheses	1.7 times faster than single GPU evaluation	Slightly slower learning time (by a few seconds) than using a single GPU	All datasets need to be in GPU memories + proper configuration of multi-GPU scheduling parameters
Hypothesis search	Multi-core CPU	Large search space	Parallel expansion of best *N* hypotheses	Up to 360x performance increase (based on IMDB dataset with parallel search and sequential evaluation)	Similar performance to sequential search	Choosing appropriate value for *N* (number of threads)

In terms of the application domain, a variation of SPILDL, known as SPILDL-ES (SPILDL for embedded systems), is proposed to facilitate high performance ILP learning in DL for systems with limited computing capabilities (i.e. embedded systems). The key contribution of SPILDL-ES, apart from inheriting SPILDL parallel hypothesis search and evaluation (using multi-core CPUs instead of GPUs), is the addition of reliability measures from critical systems, such as watchdog timers and MISD (parallel computing) architecture, with real time monitoring of operational variables to ensure reliable yet high performance of DL-based ILP learning in a harsh embedded operating environment. According to the experimental results, a threefold performance increase is achieved when parallel search and evaluation are used.

The proposed SPILDL and SPILDL-ES and their performance discussed above provide solid evidence for answering the two research questions.

RQ1: How can scalability issues of DL-based ILP learning be addressed using parallel computing?

Answer: Parallel computing addresses the scalability issues for ILP learning in DL by accelerating its two key computations (i.e. hypothesis search and hypothesis evaluation) by using multiple collaborating parallel processors with different hardware architectures.

RQ2: To what degree can parallel computing improve ILP learning in DL?

Answer: Parallel computing improves ILP learning by a huge margin, especially when parallel hypothesis search is used, resulting in speedups of up to 360 times faster than the original speed (as seen from Table 8.1).

8.2 Knowledge contribution

The work reported in the thesis makes a valuable contribution to both ILP research and ILP applications development. Concerning the former, as part of the development for the multi-GPU evaluation engine for DL hypotheses, we designed several new algorithms (Algorithm 5.1–Algorithm 5.10) for computing the results of DL operations on any parallel processor architecture (e.g. CPUs with SIMD instruction sets, multi-core CPUs, or GPUs). We also designed a novel algorithm (as illustrated in Figure 5.17) for scheduling and evaluating hypotheses on multiple GPUs, and it considers the computing capability of each GPU by probing queries to ensure more adaptive work distribution policy among GPUs.

More importantly, we developed a parallel search algorithm (Algorithm 6.1) inspired by the state of the art (the OCEL algorithm) that employs a novel approach to check hypotheses redundancy through a series of parallel reduction operations and therefore significantly reduces to negligible levels the search overhead commonly associated with parallel search algorithms. The algorithm design involves intense use of parallel computing to speed up hypothesis search by parallel expanding the best search nodes, parallel computing the final set of unique non-redundant search nodes, and parallel sorting the search nodes in the open list. These mechanisms maximize the efficiency by utilizing all the computing power

of multi-core CPUs. The algorithms outlined were demonstrated to significantly outperform their baseline counterparts.

Concerning the contribution to ILP applications development, we developed three artefacts during the course of this investigation: a novel multi-GPU evaluation engine for DL hypotheses using a heterogeneous mix of local and remote GPUs, SPILDL, and SPILDL-ES. The multi-GPU evaluation engine is capable of evaluating hypotheses on all four computing architectures: SISD (single-core CPUs), SIMD (GPUs), MISD (in reliable ILP learning), and MIMD (multi-core CPUs), as discussed in chapters 5 and 7. It is highly customizable in terms of adding and using more GPUs for hypothesis evaluation and highly configurable in terms of its scheduling policies and their parameters in order to suit many learning situations. The multi-GPU engine can be used as a standalone component with other ILP learners in DL for high-performance and scalable hypothesis evaluation, such as in combination with a genetic-based hypothesis search algorithm.

Both the SPILDL and SPILDL-ES are flexible general-purpose ILP learners in DL, by which they can run (learn) on simple machines that have single-core CPUs with no GPUs (using sequential search and sequential evaluation) as well as on powerful machines that have multi-core CPUs with multiple GPUs; in this latter configuration, the collective computing power is efficiently exploited to accelerate computations. The learning capabilities of SPILDL (including its evaluation engine) can be provided as web services ("ILP learning as a service") or as a software library, with which it can be integrated smoothly as a subsystem (component) within a larger system to provide the functionality for high-performance ILP learning in DL. For SPILDL-ES, it is meant to be used as a complete system (or a subsystem) since it handles both ILP learning in DL and monitoring of the learning hardware to ensure its stability and reliability.

The three artefacts outlined above are unique systems that are ready for use by other researchers or system developers to interface as part of their ILP systems. ILP learning for real-world applications, especially large-scale ones, was limited for many years because of its poor scalability even though it has an excellent capability for multi-relational learning. Since this work has addressed this key scalability issue with great effectiveness (by achieving large speedups) and efficiency (by using all available processors regardless of their hardware architecture), it opens the door to ILP learning for large-scale, real-world applications with a great level of performance, efficiency, and scalability that would have otherwise not been possible.

8.3 Future work

In the short term, SPILDL can be improved in terms of its DL-based ILP learning capabilities and implementations. For ILP learning in DL, more expressive DL constructs, such as string concrete roles, can be supported with operators like $stringCRole = "value"$, $stringCRole$ **contains** $"value"$, and $stringCRole$ **not contains** $"value"$, all of which can be used to build hypotheses; these constructs

further increase the expressivity for the learned models at the cost of increasing the search space complexity. Currently, both SPILDL and SPILDL-ES employ shared-memory (single machine) parallel search, which limits the computing power dedicated to the CPUs within a single machine. In addition, a single machine has an upper limit (imposed by hardware) that restricts memory in terms of its space and bandwidth, so memory space can only be increased up to a certain limit. This limitation restricts the maximum size for ILP datasets and the areas of search space that can be explored. In order to address these limitations, a version of SPILDL that employs distributed parallel search is considered as a route for our immediate future investigation.

For the implementation aspects, when evaluating a hypothesis, SPILDL feeds individual DL operators to the GPU by the CPU. This coordination introduces a (bus) communication overhead between the CPU and GPU; though it is negligible for evaluating a single hypothesis. However, it takes a considerable amount of CPU time when there is a large number of generated hypotheses that needs to be evaluated at the same time. This overhead gradually affects the total learning time; the longer the learning process is, the more accumulated effect this overhead has on the total learning time. The problem can potentially be addressed by moving all computations for hypothesis evaluation to the GPU. That change would free the CPU to do other useful work and therefore increase the efficiency by allowing the overhead time gap to be used for GPU computation instead of waiting. For multi-GPU evaluation, a dynamic multi-GPU scheduling can be designed, whereby GPUs can communicate with one another directly either locally (through the shared bus) or remotely via the network interface (through the PCI bus), with minimal or potentially zero CPU intervention for purpose of work distribution and load balancing. This increases the efficiency for multi-GPU computations by significantly reducing the communication and coordination overheads, potentially to negligible levels. The aforementioned implementation improvements can help to extract more performance and consequently contribute to shorter learning times. Our immediate future work will include such improvements.

In the medium term, case studies for using SPILDL in a particular application area, such as analysing car accident patterns or ILP-based product recommendations in e-commerce, can be developed. The effectiveness of the system and SPILDL can then be evaluated in an application (or user) context. In an application setting with SPILDL, there are three options to deal with ILP datasets. The first is to load the DL dataset from an OWL file, as in the current implementation. The second option is to use a SPARQL query to retrieve the learning dataset from a remote OWL source. SPILDL connects to a user-defined remote OWL data repository (e.g. DBpedia) to retrieve input dataset from the query results. In the third option, SPILDL matrices can be updated directly from the incoming stream of data (e.g. sensor data) by either adding new assertions or updating the existing ones. The direct access of matrices can also be used for assertions related to a robot experience, with which a robot's updated knowledge base (accumulated experiences) helps it to make real time decisions from an ILP model. The first and second

option are already supported by the DL-Learner; however, unlike SPILDL the DL-Learner cannot handle large DL datasets directly. Therefore, its support for the second option is limited in terms of dataset size. The third option allows the knowledge base (matrices) to be updated quickly to respond to live environment changes, which is particularly useful for robotic/agent applications. Designing and implementing SPILDL based systems with live data (i.e. the second and third option) is our medium term plan of future work.

In the long run, the expressiveness of the SPILDL learned models can be expanded to include models expressed in horn clauses (similar to classical ILP learners), potentially including FOL functions as well. This expansion of models immensely extends the contribution beyond DL-based ILP learners. A key advantage for ILP learners in horn clauses is their highly expressive hypothesis language. However, parallelizing hypothesis evaluation for horn clauses is a challenging issue. A common approach to address this problem is to move hypothesis evaluation to database systems, where a predicate is mapped to a database table and its variables to table columns. This solution improves the scalability for evaluating hypotheses in horn clauses; however, evaluating a single hypothesis introduces an overhead that may potentially cancel out the benefits of parallelizing the evaluation. The GPU-based implementations can significantly reduce this overhead, though consideration must be taken for the nature of horn clauses and the computing architecture for GPUs (i.e. SIMD) to achieve as much performance as possible. Designing a variation of SPILDL for horn clauses that combines both parallel hypothesis search and evaluation (i.e. SPILFOL (Scalable and Parallel Inductive Learner in First Order Logic)) is our long-term future work.

SPILDL can also potentially be combined with other high-performance scalable machine learners in the AI literature, such as GPU-accelerated neural networks to achieve scalable and powerful hybrid machine learners. A key issue with these hybrid learners is their limited scalability resulting from the combination with a (non-scalable) ILP learner which constrains their learning potential. Combining a scalable ILP learner with a scalable machine learning algorithm, such as neural networks, will not only facilitate learning from massive datasets directly but also will improve the ILP learning portion for theses hybrid learners. Better ILP models can be learned since the entire ILP dataset can be used directly instead of relying on a small sample of ILP data, which may miss critical information about the learning problem.

References

Abburu, S. (2012) ‘A survey on ontology reasoners and comparison’, *International Journal of Computer Applications,* 57(17), pp. 33-39.

Alonso, E. and Kudenko, D. (2000) ‘Logic-based multi-agent systems for conflict simulations’, *Proceedings from the 5th UK Workshop United Kingdom Multicentre Acamprosate Study (UKMAS’ 00),* Oxford.

Baader, F. et al. (2017) *An introduction to description logic.* Cambridge: Cambridge University Press.

Baader, F., Lutz, C. and Suntisrivaraporn, B. (2006) ‘Efficient reasoning in EL+*’, Proceedings of the 2006 International Workshop on Description Logics (DL2006),* Lake District, pp. 15-26.

Barney, B. (2010) *Introduction to parallel computing.* Livermore: Lawrence Livermore National Laboratory.

Blockeel, H. and de Raedt, L. (1998) ‘Top-down induction of first-order logical decision trees’, *Artificial Intelligence,* 101(1-2), pp. 285-297.

Blockeel, H. et al. (2000) ‘Executing query packs in ILP’, *Proceedings of the 10th International Conference on Inductive Logic Programming (ILP2000),* London, July 24-27, pp. 60-77.

Bratko, I. and Muggleton, S. (1995) ‘Applications of inductive logic programming’, *Communications of the ACM,* 38(11), pp. 65-70.

Bühmann, L., Lehmann, J. and Westphal, P. (2016) ‘DL-Learner - a framework for inductive learning on the Semantic Web’, *Web Semantics: Science, Services and Agents on the World Wide Web,* 39, pp. 15-24.

Calvanese, D. et al. (2017) ‘Ontop: Answering SPARQL queries over relational databases’, *Semantic Web,* 8(3), pp. 471-487.

Chantrapornchai, C. and Choksuchat, C. (2018) ‘TripleID-Q: RDF Query processing framework using GPU’, *IEEE Transactions on Parallel and Distributed Systems,* 29(9), pp. 2121-2135.

De Giacomo, G. et al. (2012) ‘Mastro: A reasoner for effective ontology-based data access’, *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE2012)*, Manchester, pp. 71-81.

De Raedt, L. et al. (2015), ‘Inducing probabilistic relational rules from probabilistic examples’, *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*, Buenos Aires, pp. 1835-1843.

Dean, J. and Ghemawat, S. (2008) ‘MapReduce: simplified data processing on large clusters’, *Communications of the ACM*, 51(1), pp. 107-113.

Debnath, A. K. et al. (1991) ‘Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity’, *Journal of Medicinal Chemistry*, 34(2), pp. 786-797.

Dehaspe, L. and Toivonen, H. (1999) ‘Discovery of frequent DATALOG patterns’, *Data Mining and Knowledge Discovery*, 3(1), pp. 7-36.

DiMaio, F. and Shavlik, J. (2004) ‘Learning an approximation to inductive logic programming clause evaluation’, *Proceedings of the 14th International Conference on Inductive Logic Programming (ILP2004)*, Porto, pp. 80-97.

Džeroski, S., de Raedt, L. and Driessens, K. (2001) ‘Relational reinforcement learning’, *Machine Learning*, 43, pp. 7-52.

Fanizzi, N., d’Amato, C. and Esposito, F. (2008) ‘DL-FOIL concept learning in description logics’, *Proceedings of the 18th International Conference on Inductive Logic Programming (ILP2008)*, Prague, pp. 107-121.

Feng, C. (1991) ‘Inducing temporal fault diagnostic rules from a qualitative model’ , *Proceedings of the 8th International Conference on Machine Learning (ML'91)*, Evanston, June 1, pp. 403-406.

Flynn, M. J. (1972) ‘Some computer organizations and their effectiveness’, *IEEE Transactions on Computers*, C-21(9), pp. 948-960.

Fonseca, N. A., Silva, F. and Camacho, R. (2005) ‘Strategies to parallelize ILP systems’, *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP2005)*, pp. 136-153.

Fonseca, N. A. et al. (2009) ‘Parallel ILP for distributed-memory architectures’, *Machine Learning*, 74(3), pp. 257-279.

Fukunaga, A. et al. (2017) ‘A survey of parallel A’, *arXiv*, 1708.05296. Available at: https://arxiv.org/pdf/1708.05296v1.pdf (Accessed: 20 October 2020).

Fukunaga, A. et al. (2018) ‘Parallel A* for State-Space Search’, Handbook of Parallel Constraint Reasoning, pp. 419-455.

Glimm, B. et al. (2014) ‘HermiT: an OWL 2 reasoner’, *Journal of Automated Reasoning*, 53(3), pp. 245-269.

Glimm, B., Horrocks, I. and Sattler, U. (2008) ‘Deciding SHOQ⊓ Knowledge base consistency using alternating automata’, *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, Dresden.

Iannone, L., Palmisano, I. and Fanizzi, N. (2007) ‘An algorithm based on counterfactuals for concept learning in the Semantic Web’, *Applied Intelligence*, 26(2), pp. 139-159.

Kazakov, Y., Krötzsch, M. and Simancik, F. (2012) ‘ELK reasoner: architecture and evaluation’, *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE2012)*, Manchester, pp. 100-111.

Khronos Group (no date). *OpenCL overview.* Available at: https://www.khronos.org/opencl/ (Accessed 20 October 2020).

Konstantopoulos, S. (2007) ‘A data-parallel version of Aleph’, *arXiv*, 0708.1527. Available at: https://arxiv.org/pdf/0708.1527.pdf (Accessed 20 October 2020).

Lehmann, J. (2010) ‘Learning OWL class expressions’, PhD Thesis, University of Leipzig, Leipzig.

Law, M., Russo, A. and Broda, K. (2018a) ‘The complexity and generality of learning answer set programs’, *Artificial Intelligence*, 259, pp. 110-146.

Law, M., Russo, A. and Broda, K. (2018b) ‘Inductive Learning of Answer Set Programs from Noisy Examples’, arXiv, 1808.08441. Available at: https://arxiv.org/pdf/1808.08441.pdf (Accessed 9 March 2021).

Law, M., Russo, A. and Broda, K. (2019) ‘Logic-Based Learning of Answer Set Programs’, *Reasoning Web. Explainable Artificial Intelligence*, pp. 196-231.

Law, M. et al. (2020) ‘FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria’, *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(3), pp. 2877-2885.

Martínez-Angeles, C. A. et al. (2016) ‘Relational learning with GPUs: accelerating rule coverage’, *International Journal of Parallel Programming*, 44(3), pp. 663-685.

Meissner, A. (2009) ‘A simple parallel reasoning system for the ALC description logic’, *Proceedings of the 1st International Conference on Computational Collective Intelligence (ICCCI 2009)*, Wroclaw, October 5-7, pp. 413-424.

Microsoft (2018) *DirectCompute.* Available at: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader (Accessed 20 October 2020).

Mitchell, T. M. (1997) *Machine learning.* New York: McGraw Hill.

Mooney, R. J. and Califf, M. E. (1995) ‘Induction of first-order decision lists: results on learning the past tense of english verbs’, *The Journal of Artificial Intelligence Research*, 3, pp. 1-24.

Mozart System (2018) *The Mozart programming system.* Available at: http://mozart.github.io/ (Accessed 20 October 2020).

Muggleton, S. (1995) ‘Inverse entailment and progol’, *New Generation Computing,* 13(3), pp. 245-286.

Muggleton, S. and de Raedt, L. (1994) ‘Inductive logic programming: theory and methods’, *The Journal of Logic Programming,* 19-20, pp. 629-679.

Muggleton, S. and Feng, C. (1990) ‘Efficient induction of logic programs’, *Proceedings from the First Conference on Algorithmic Learning Theory,* Tokyo, pp. 368-381.

Nenov, Y. et al. (2015) ‘RDFox: a highly-scalable RDF store’, *The Semantic Web (ISWC 2015),* pp. 3-20.

Nezhad, A. T. and Muggleton, S. (2002) ‘A genetic algorithms approach to ILP’,*Proceedings of the 12th International Conference on Inductive Logic Programming (ILP2002)*, Sydney, July 9-11, pp. 285-300.

Nishiyama, H. and Ohwada, H. (2015) ‘Yet another parallel hypothesis search for inverse entailment’, *Proceedings of the 25th International Conference on Inductive Logic Programming (ILP2015),* Kyoto, August 20-22, pp. 86-94.

Nvidia (2007) *Nvidia CUDA programming guide.* Available at: http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf (Accessed 20 October 2020).

Nvidia (2009) *NVIDIA’s next generation CUDA compute architecture.* Available at: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (Accessed 20 October 2020).

Nvidia (2012) *NVIDIA GPUDirect technology.* Available at: http://developer.download.nvidia.com/devzone/devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf (Accessed 20 October 2020).

Nvidia (no date a) *Cuda zone.* Available at: https://developer.nvidia.com/cuda-zone (Accessed 20 October 2020).

Nvidia (no date b) *GPU applications catalog.* [Online] Available at: https://www.nvidia.com/en-us/gpu-accelerated-applications/ (Accessed 20 October 2020).

Ohwada, H. and Mizoguchi, F. (1999) 'Parallel execution for speeding up inductive logic programming systems', *Proceedings of the 2nd International Conference on Discovery Science (DS 1999)*, Tokyo, December 6-8, pp. 277-286.

Ohwada, H., Nishiyama, H. and Mizoguchi, F. (2000) 'Concurrent execution of optimal hypothesis search for inverse entailment', *Proceedings of the 10th International Conference on Inductive Logic Programming (ILP2000)*, London, July 24-27, pp. 165-173.

OpenGL Wiki contributors (2019) *Compute shader.* Available at: https://www.khronos.org/opengl/wiki/Compute_Shader (Accessed 20 October 2020).

OpenMP (no date) *OpenMP.* Available at: www.openmp.org (Accessed 20 October 2020).

OpenMPI (2020) *Open MPI.* Available at: https://www.open-mpi.org/ (Accessed 15 May 2020).

Petnga, L. and Austin, M. (2016) 'An ontological framework for knowledge modeling and decision support in cyber-physical systems', *Advanced Engineering Informatics,* 30(1), pp. 77-94.

Qomariyah, N. N. and Kazakov, D. (2017) 'Learning from ordinal data with inductive logic programming in description logic', *Proceedings of the 27th International Conference on Inductive Logic Programming (ILP2017)*, Orléans, September 4, pp. 38-50.

Quinlan, J. R. (1986) 'Induction of decision trees', *Machine Learning,* 1(1), pp. 81-106.

Quinlan, J. R. (1990) 'Learning logical definitions from relations', *Machine Learning,* 5(3), pp. 239-266.

Serrurier, M. et al. (2007) 'Learning fuzzy rules with their implication operators', *Data & Knowledge Engineering,* 60(1), pp. 71-89.

Sirin, E. et al. (2007) 'Pellet: a practical OWL-DL reasoner', *Journal of Web Semantics,* 5(2), pp. 51-53.

Srinivasan, A. (1999) ‘A study of two sampling methods for analysing large datasets with ILP’, *Data Mining and Knowledge Discovery,* 3(1), pp. 95-123.

Srinivasan, A. (2007) *The Aleph manual.* Available at: https://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html [Accessed 20 October 2020].

Srinivasan, A., Faruquie, T. A. and Joshi, S. (2010) ‘Exact data parallel computation for very large ILP datasets’, *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP2010)*, Florence

Steigmiller, A., Liebig, T. and Glimm, B. (2014) ‘Konclude: system description’, *Journal of Web Semantics*, 27-28, pp. 78-85.

Tsarkov, D. and Horrocks, I. (2006) ‘FaCT++ description logic reasoner: system description’, in Furbach, U. and Shankar, N. (eds.) *IJCAR 2006: Automated Reasoning*. Berlin: Springer, pp. 292-297.

Tuckey, D., Broda, K. and Russo, A. (2020) ‘Towards Structure Learning under the Credal Semantics’, *The 36th International Conference on Logic Programming (ICLP2020) Workshops*, Rende, September 18-25.

University of Edinburgh (2020) *GPGPU computing.* Available at: http://computing.help.inf.ed.ac.uk/gpgpu-computing (Accessed 20 October 2020).

Wu, K. and Haarslev, V. (2012) ‘A parallel reasoner for the description logic ALC’, *Proceedings of the 25th International Workshop on Description Logics (DL2012)*, Rome, pp. 378-388.

Zeng, Q., Patel, J. M. and Page, D. (2014) ‘QuickFOIL: scalable inductive logic programming’, *Proceedings of the VLDB Endowment (PVLDB)*, 8(3), pp. 197-208.

Zhou, Y. and Zeng, J. (2015) ‘Massively parallel A* search on a GPU’, *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*, Austin pp. 1248-1254.